12-2021

COMPARATIVE ANALYSIS OF KMER COUNTING AND ESTIMATION TOOLS

Ankitha Vejandla
University of Nebraska - Lincoln, avejandla2@unl.edu

COMPARATIVE ANALYSIS OF KMER COUNTING AND ESTIMATION TOOLS

by

Ankitha Vejandla

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Jitender S. Deogun

Lincoln, Nebraska

December, 2021

COMPARATIVE ANALYSIS OF KMER COUNTING AND ESTIMATION TOOLS

Ankitha Vejandla, M.S.

University of Nebraska, 2021

Advisor: Jitender S. Deogun

The rapid development of next-generation sequencing (NGS) technologies for determining the sequence of DNA has revolutionized genome research in the recent years. *De novo* assemblers are the most commonly used tools to perform genome assembly. Most of the assemblers use de Bruijn graphs that break the sequenced reads into smaller sequences (sub-strings), called kmers, where k denotes the length of the sub strings. The kmer counting and analysis of kmer frequency distribution are important in genome assembly. The main goal of this research is to provide a detailed analysis of the performance of different kmer counting and estimation tools that are currently available. This helps the bioinformatics researchers to make a good decision in choosing accurate and efficient tools for genome assemblers.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Deoxyribonucleic acid (DNA) is the basis of existence of life on the planet Earth. A DNA molecule is made up of four nucleotides, adenine, guanine, cytosine, and thymine, depicted as A,G,C and T, respectively, and are also known as the building blocks of a DNA molecule. Two DNA strands are coiled together and nucleotides are arranged pair wise, which tend to stick to their compliment nucleotides. A's compliment is T and C's compliment is G and vice versa. This arrangement allows the reconstruction of one strand using the compliment strand's nucleotide sequence as the template. Figure1.1 shows the structure of a DNA molecule.

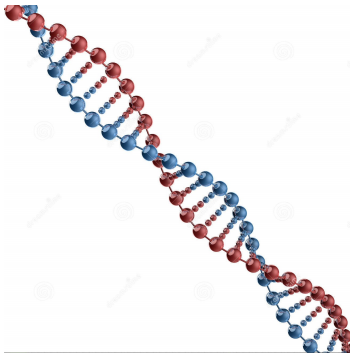

Figure 1.1: DNA Molecule

The order in which these nucleotides are present in a DNA molecule provides a lot of information about the biological behavior of an organism. The process of determining the arrangement of the nucleotides is known as DNA sequencing. Knowledge of DNA sequences has become very useful in various fields such as medical diagnosis, biotechnology, forensics, agriculture, virology, archaeology, anthropology and so on.

Examples of how DNA sequences are used as follows:

Microbiology: Information provided by DNA sequencing helps researchers to identify the genetic changes that can be associated with diseases, which is very useful to identify the type of drugs to cure diseases.

Forensics: DNA sequencing is used along with DNA profiling, which is a technique used to identify individuals with the help of their DNA characteristics. These two techniques are helpful in forensics investigations to identify an individual based on fingerprints or any other available DNA sample.

Agriculture: DNA sequencing information of microorganisms has proven very useful for the agriculturists to make use of the specific kinds of bacteria that can provide nutritious values for the crops and food plants.

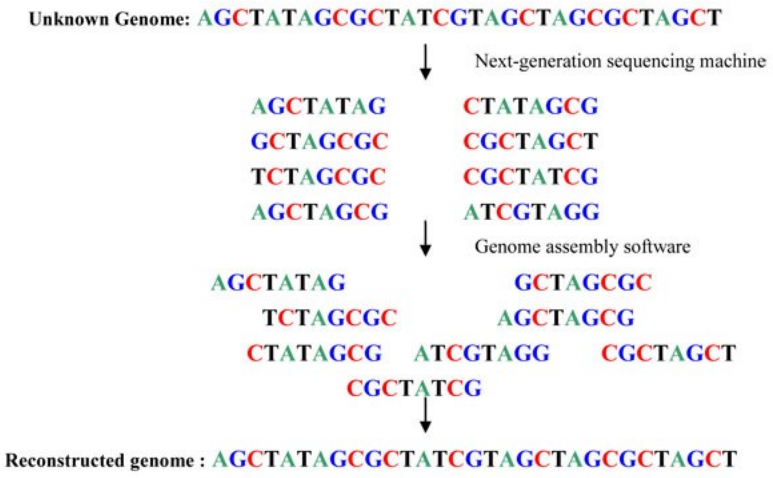

Figure 1.2: Genome Assembly

In genome sequencing, as shown in Figure 1.2, a genomic sequence is subjected to fragmentation of the reads. This results in a large quantity of sequencing reads, which must then be assembled to reconstruct the original sequence. Most common methods used to accomplish this assembly are sanger sequencing and next generation sequencing techniques.

1.1.1 Sanger Sequencing

Sanger sequencing technique was developed by Fredrick Sanger in the year 1977, for which he was awarded a Nobel Prize in the year 1980. This was the first method devised to sequence DNA [34]. In this method, first the two DNA strands are separated and the strand to be sequenced is copied using chemically altered bases. The altered bases stop the copying process each time a particular nucleotide is recognized in the DNA sequence, which is growing. This process is repetitively carried out for all the four nucleotides and the resulting fragments are arranged to obtain the final DNA sequence. Sanger sequencing is known as the gold standard for DNA sequenc-

ing. One of the biggest drawbacks of Sanger sequencing is that it is not possible to sequence very long, hundred million bases of a genomic sequence. The next generation technologies provided many efficient sequencing frameworks that overcame this drawback.

1.1.2 Next Generation Sequencing Technologies

Next Generation Sequencing (NGS) technologies[2] can produce a huge amount of sequence data in a significantly short amount of time. The powerful and flexible nature of NGS has made it an indispensable tool for a broad spectrum of biological sciences. These technologies produce a large amount of short reads. Several genome assemblers are available for these types of data. *De novo* assemblers are one of the commonly used tools to perform genome assembly.

The two major approaches in *de novo* assemblers are based on Overlap Graphs (OLG) and De Bruijn Graphs (DBG).

1.1.2.1 Overlap Graph Assembler

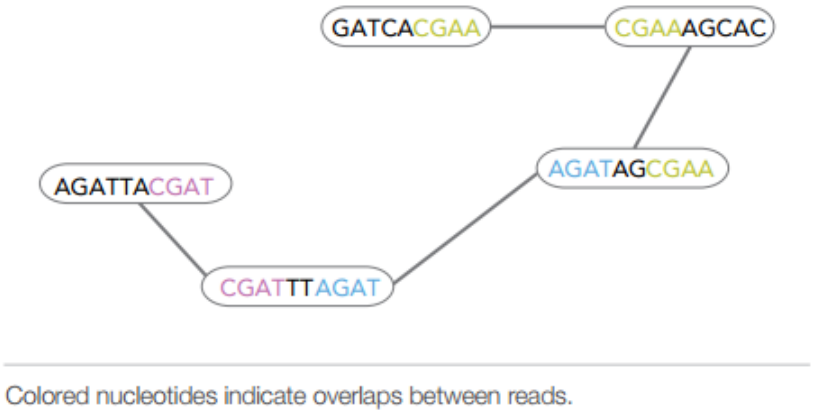

Figure 1.3: Overlap Graphs

Some of the established assemblers follow the overlap-layout-consensus paradigm. They compute pairwise overlaps between the reads and plot a graph with this information. Nodes of the graph corresponds to a read and the edges denotes an overlap between two nodes or reads as shown in figure 1.3. This method is used to compute a layout of reads and a sequence of contigs (continuous sequences). For this method to perform well, there must be a significant overlap between the reads. This approach is traditional and computationally intensive. Even for an assembly of a simple short-genome, millions of reads need to be assembled, making the graph extremely complex.

1.1.2.2 De Bruijn Graph Assembler

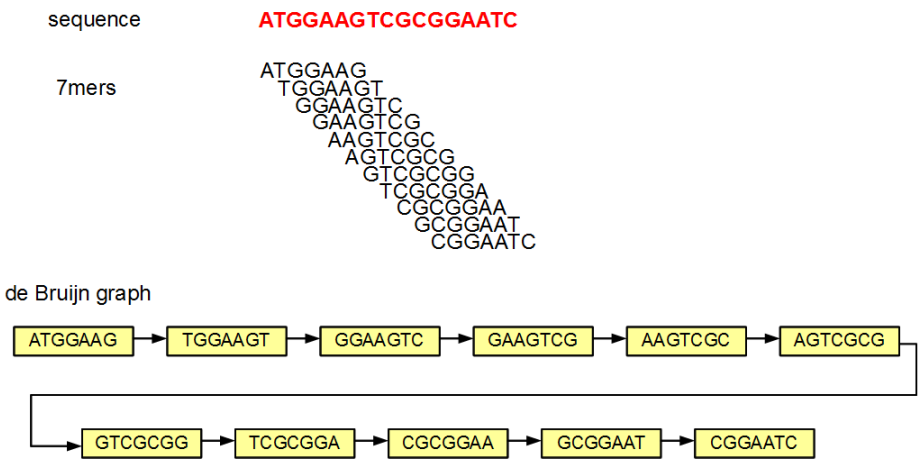

Figure 1.4: De Bruijn Graphs

Since overlap graphs do not scale with growing reads, most of the assemblers for NGS use de Bruijn graphs. They break the reads into smaller sequences, called kmers, where k denotes the length of sub strings of DNA sequences. The nodes of this graph are of length k and edges of length k-1. The graph is formed out of kmers and not the actual reads. Figure 1.4 shows an example of a de Bruijn graph with k = 7.

This technique reduces the dataset into kmers. The most important parameter for graph construction, k is determined by various factors, such as read length and error rate. The quality of assembly depends up on the value of k. Best value of k, is often determined by testing the reads with a range of values.

Since *de novo* assemblers use de Bruijn graphs, kmer counting has become one of the important tools in genome assembly process.

1.2 Scope of research

The main goal of this research is to provide a detailed analysis of the performance of various kmer counting and estimation tools that are currently available. This helps the bioinformatics researchers to make a good decision to choose efficient and accurate tools.

Kmer counting is the process of identifying strings of length k and their frequencies in a genomic sequence. Some of the kmer counting tools do not provide a list of unique kmers (frequency=1) as they are often considered as erroneous.

Kmer estimation in the process of providing an estimate of the number of distinct kmers, kmers of different frequencies and a frequency histogram.

Kmer counting and estimation has gained a lot of importance in the field of bioinformatics recently. A lot of research has been done in this space comparing the kmer tools. Our research provides a comparison of 13 kmer counting tools and 5 kmer estimation tools. The kmer counting tools evaluated are BFCounter [21], Turtle [30], JellyFish [20], DSK [29], KAnalyze [5], KMC2 [10], MSPKmerCounter [15], Khmer [38], Tallymer [13], KCMBT [18], Squeakr [26], GenomeTester4 [12], Gerbil [11]. Kmer estimation tools are KmerStream [22], Kmerlight [36], KmerGenie [8], ntCard [23] and KmerEstimate [6]. Experiments were conducted with 4 different data sets of variable

sizes and compared the performance, which is the time of execution and memory used for execution. Also analyzed the algorithms of each of these tools and their implementation techniques. Our research is the first to provide a performance comparison between the above listed kmer counting and estimation tools.

Chapter 2

Overview of Kmer Counting Tools

Table 2.1: Kmer Counting Tools

Approach	Tools
LCP - Interval Tree Type	Tallymer
Bloom Filter Based	BFCounter
	Turtle
Count Min Sketch	Khmer
Partition Based	DSK
	KAnalyze
Lock Free Queues	Jellyfish
Minimizers	MSPKmerCounter
	KMC2
Multiple Burst Trees	KCMBT
Counting Quotient Filter Based	Squeakr
GPU Computing	Gerbil
SAC	GenomeTester4

Kmer counting tools are grouped into 10 different categories based on their algorithmic approach, as shown in Table 2.1. We provide a brief overview of each of these tools, their algorithms and methods.

2.1 Bloom Filter Based

Bloom filter is a probabilistic data structure, which is used to find whether a given element is part of a data set, while managing space efficiently [7]. BFCounter and Turtle are the two bloom filter based tools.

2.1.1 BFCounter

BFCounter is a bloom filter based kmer counting algorithm that provides the list of kmers which appear more than once in a DNA sequence data [21]. BFCounter performs kmer counting in multiple passes. In the first pass, a bit array is used to which all the kmers are mapped using multiple hash functions. In the second pass, all the kmers that are already mapped in the first pass are revisited with a hash table that stores the count of kmers. In the third pass, all the unique kmers that are added to the hash table in the second pass are removed and all the non-unique kmers are stored and displayed as the output.

The total number of kmers expected from the input sequence of the size n can be calculated by using the formula n-k+1. Then, the number of unique kmers can be obtained by subtracting the number of non-unique kmers obtained by BFCounter as described above from the number of total kmers.

2.1.2 Turtle

Turtle is also a bloom filter based kmer counting algorithm. Although both the algorithms use bloom filter as the underlying data structure, there is a significant difference in the implementation of these tools. Turtle performs kmer counting in two passes. First pass uses a pattern blocked bloom filter [28]. The main difference between a general bloom filter [7] and a pattern blocked bloom filter is that, in a

general bloom filter, there is no control over the indices of the bit array that are generated by the multiple hash functions, where as in a pattern blocked bloom filter, some control is exercised on the indices generated by the hash functions, which ensures that the indices are not very far from each other and reduces the time and space utilized by the tool in the first pass. In the second pass a technique called sorting and compaction (SAC) [30] is performed, where each kmer from the input data is checked for its presence in the bloom filter from the first pass and if present, it is then added to an array. This process goes on till the array reaches a threshold, after which all the elements that appear multiple times are sorted and their count is increased and then removed from the array by creating space for the other kmers. This process saves time and space as performing sort operation on an array is less costly compared to a hashmap.

Turtle also outputs only non-unique kmers as BFCounter. We can use the same technique mentioned in the above section to find the count of unique kmers from Turtle.

2.2 Lock Free Queues

Queue, as the name suggests is a first in first out (FIFO) data structure. Lock free queues are designed in a way that enqueue and dequeue operations can be performed simultaneously [14], unlike locked queues which can perform one operation at a time.

2.2.1 JellyFish

Jellyfish is a kmer counting tool that provides the count of all the kmers in a DNA sequence data including the ones with frequency 1. The algorithm is designed to make use of CAS (compare and swap operation) [14] available in most of the modern

CPUs. A light hash function is designed and used in this algorithm. CAS is an atomic instruction mostly used in multi-threading for synchronization of write operations performed by the program. CAS function consists of three arguments, location, old value and new value. It goes to the location where a write operation is supposed to be performed and compares the old value with the value that is present in the location, if there is a match then it updates the old value with the third parameter new value and returns old value on success [20]. It will not perform write operation if the values do not match and hence CAS fails [20].

Jellyfish algorithm is performed in two elaborate steps. The first step consists of key mapping, where a hash function is used to generate a key-value pair with kmer as the key and a value v, which is the location of the kmer in the hashmap. CAS is used to check if the location is empty or occupied with old value as kmer. If it returns kmer then a successful insertion is made to the hashmap. If the position is occupied, then a reprobe function is used to provide a different position to the kmer [20]. The second step of the algorithm is used to increment the count of kmers. In this step, a CAS function is used to check if the kmer is already present in the hash table and if it is, then value of the kmer (key) is incremented. For efficient memory usage, when the hashmap is almost full, data is written to the hard disk and hash table is emptied [20].

2.3 Partitioning Technique

DSK and KAnalyze use partitioning technique to minimize the memory required for kmer counting. These tools divide the data into partitions and perform computations on the partitioned data.

2.3.1 DSK

DSK (Disk Streaming of Kmers) is a Kmer counting algorithm developed specifically to perform kmer counting for large datasets with minimal memory usage. DSK is the first kmer counting tool to count 27mers of human genome dataset using 4GB memory, where tools like BFCounter and Jellyfish used 56GB and 70GB respectively[29].

The algorithm minimizes the memory and disk space used by partitioning the data and having one partition in memory at any time and rest of the partitions on the disk. It takes maximum memory and disk space as input from users and performs kmer counting. The algorithm calculates the number of iterations that it needs to partition the kmers and to write them to the disk and so on [29]. This is calculated from the user inputs. In the next step, a hash function is used that can decide which partition that the kmer has to be a part of. In the final step, one partition is processed at a time and the number of kmers are written to a final file on the disk, where the kmer count from different partitions is synced and the final count is presented.

2.3.2 KAnalyze

KAnalyze is a kmer counting tool, which is implemented in Java unlike the other tools that are implemented in C++. The algorithm works in two steps, split and merge. Data is split into several subsets and at any point of time only one subset is loaded into the memory [5]. Each subset is then sorted with kmers and their count. This information is then passed onto a merge step, where the sorted subsets are merged to a final output file, which at the end consists of all the kmers and their frequencies. With this split and merge approach, KAnalyze uses a limited amount of memory at any point of time thus utilizing less space and the sorting step often uses an algorithm with best time complexity. KAnalyze also has an efficient run time.

2.4 Minimizers

2.4.1 MSPKmerCounter

MSPKmerCounter uses a technique called Minimum Substring Partitioning [16] to provide the exact count of kmers from a DNA sequence data set. The algorithm finds the sub strings of the sequence that appear in the maximum number of distinct kmers of size p where p<=k. These substrings, which are called minimizers are then used to partition the kmers that contain these minimizers into one partition. The kmers that contain the minimizers are known as super kmers[15]. In the next step, all the super kmers are then visited and processed in such a way that the kmer and their respective counts are stored in a hashmap, which is the output provided by the algorithm.

2.4.2 KMC2

KMC2 is a kmer counting tool that follows two steps: distribution and sorting to provide a list of kmers and their respective frequencies. This tool makes use of a concept called minimizers that is also used in MSPKmerCounter, but replacing the original minimizers with signatures (subset of minimizers). Using (k, x)-mers also known as signatures allows for a parallel overall architecture. Usage of signatures also improves the execution time and memory consumption of the tool. In the distribution step, minimizers, which can be defined as an m-mer of a kmer such that there cannot be another lexicographically smaller m-mer present for that particular kmer [10]. This concept of minimizers has been modified to signatures, such that each m-mer occupies less size and the value m is moderate. There is also a condition that the signatures cannot start with ‘AAA’ or ‘ACA’ and ‘AA’ is not contained in the kmer except at the beginning [10]. In the distribution step, each read is examined to find the overlapping

regions called super kmers and these are sent to bins, related to signatures. In the second stage, each of these files written to the disk from the above step are extracted to form (k,x)-mers where x= 0,1,2,3...x (any small x). These (k,x)-mers are sorted and final statistics of kmer counts are then stored in a compact binary form.

2.5 Count Min Sketch

Count-min sketch is a probabilistic sublinear space data structure, which is used to find frequent items, quantiles, etc in a data stream [9].

2.5.1 Khmer

Khmer is a kmer counting tool that uses count-min sketch data structure to find the exact count of kmers in a DNA sequencing dataset. The implementation of this tool is similar to that of bloomfilter (BFCounter). However, instead of one hash table with multiple hash functions, there are multiple hash tables and instead of 1 bit counters, 8 bit counters are used to store the frequency of a particular kmer. The algorithm has three steps. The first step is to create hash tables depending on the size of the dataset. In the second step each kmer is hashed to multiple hash tables and the result of hash function depends on the size of the hash tables [38]. Once the position of a kmer is determined, the counter at that position is incremented. After all the kmers are hashed, for each kmer all the hash table counters are retrieved and the closest value is the count of that kmer. Due to a large number of kmers and limited hash tables, collisions are possible [27]. Since, the output is always the smallest value amongst all the counters, which almost minimizes the false kmer count [38].

2.6 Enhanced Suffix Array

2.6.1 Tallymer

Tallymer is a memory efficient solution for counting kmers and indexing large data sets [13]. Tallymer uses suffix arrays and longest common prefix (lcp) interval tables, which are together known as enhanced suffix arrays [1]. The idea behind the enhanced suffix arrays is that a sequence S of DNA sequencing data set is moved to an array and all the kmers (sub strings of length k) are then separated by a '$_r'. Each kmer has a prefix and a suffix. The array is then ordered based on a precedence given to A, C, T and G as suffixes lexicographically. A longest common prefix table maintained, is an integer array and contains the lengths of longest common prefixes. An lcp interval tree is then constructed and kmer counting is performed. With the help of enhanced suffix arrays and lcp trees, Tallymer processes parts of the data at any point of time, which uses less RAM resources and hence memory efficient [13].

2.7 Multiple Burst Trees

2.7.1 KCMBT

KCMBT is a kmer counting algorithm that uses Burst Trees [35], which is a fast and efficient data structure for string keys. The algorithm works in three phases. In the first phase, all the sequences are visited and are split into (k+x)-mers, where $0 \leq x \leq 3$. Each (k+x)-mer is then compared to its canonical (lexicographically smaller one between the original kmer and it's reverse compliment) form and a (k+x)-mer which is lexicographically smaller is selected and is stored in its corresponding tree. The (k+x)-mers at this point are stored in a buffer and when the buffer is full, all the

(k+x)-mers are inserted into memory at the same time saving computational time. In the second phase, all the (k+x)-mers are traversed and split into kmers, which are then inserted into trees of similar prefix kmers. In the last step, all the trees are traversed and kmers with their frequencies are obtained. All the trees used in this algorithm are burst trees, which are faster than the regular binary search trees and use lesser memory.

2.8 Counting Quotient Filter Based

2.8.1 Squeakr

Squeakr stands for Simple Quotient Filter based Exact and Approximate Kmer Representation [26]. As the name suggests, Squeakr has both an approximate and an exact kmer counting approaches - approximate is called Squeakr and exact is Squeakr-exact. For the purpose of this analysis, Squeakr the approximate tool is being considered, which is a in-memory kmer approximation approach [19].Squeakr uses Counting Quotient Filter [25] data structure to provide an approximate count of kmers. To efficiently count the number of kmers, Squeakr uses a multi-threaded approach, where the input file is divided into multiple chunks and each thread works on one piece at a time.The algorithm uses Murmurhash function to generate p bit value, using which the kmer is inserted in to a thread safe CQF. The thread then locks a particular section of CQF until the kmer is stored in the queue. To address the issue of multiple threads attempting to acquire lock at the same time, each thread is given it's own local CQF. When the global CQF is locked by a different thread, the kmer and it's counter are both written to a local CQF and all the kmers in the local CQF are then moved to global CQF once the local CQF is full. This way, there is no waiting around

time for the threads and also reduces the processing time as the data from local lock free CQF is moved to global CQF only when the local is full. CQF uses an encoding scheme, which enables it to maintain variable sized counters that stores the remainder (hash value p is divided into a quotient q and a remainder r, where p=q+r) instead of the actual count, which is memory efficient [26].

2.9 GPU Counting

2.9.1 Gerbil

Gerbil is a kmer counting tool that uses a two-disk approach that is similar to those of most contemporary kmer counting tools [11]. The algorithm is split into two phases - distribution and counting. In the distribution phase input files are split into multiple temporary files by reader threads. It then creates minimizers (substring of a kmer of length m where m<k) similar to that of KMC2, which is done by a set of splitter threads. The data into temporary files is split into super-mers, strings of max length in each file. A writer thread then writes all the super-mers to a temporary file on the disk. In the counting phase, a reader thread reads the temporary files and places them in main memory. Splitter threads then split the super-mers in the temporary files into kmers and write to a set of hash tables that uses parthash function to assign key value pairs. A writer thread then combines the hash tables and writes final kmer counts to the disk. In its graphics processing unit (GPU) implementation, the second phase is performed on the GPU side with proper load balancing between GPU and CPU [19].

2.10 Sort and Count

2.10.1 GenomeTester4

GenomeTester4 is a kmer counting tool that performs the counting using GListMaker program. In the first step, the sequences are read using the glistmaker program and are placed into multiple temporary arrays using multiple threads. The arrays are then sorted and kmers that are adjacent are counted during collation phase [12]. The temporary arrays are then removed and kmer counts are then provided as output.

Chapter 3

Overview of Kmer Estimation Tools

Table 3.1: Kmer Estimation Tools

Approach	Tools
Streaming Algorithms	KmerStream
	Kmerlight
	ntCard
	KmerEstimate
Sampling Algorithms	KmerGenie

Kmer estimation tools provide an estimate of count of unique kmers, distinct kmers, total number of kmers and abundance histograms. We compared the performance (RAM usage and execution time) between the estimation tools and a count of distinct kmers to that of DSK, which is our benchmark data. A brief description of the implementation techniques and performance results of the tools listed in Table 3.1 is as follows.

3.1 Streaming Algorithms

3.1.1 KmerStream

KmerStream is a kmer estimation algorithm, which provides an estimate of number of distinct kmers, number of unique kmers and total number of kmers present in DNA sequence data. The algorithm is designed to use a list of arrays as a data structure each of size R, which is determined by the error component that can be given as input to the algorithm. Each array indices can take values from 0 to 3 (2-bit number) and depending on the binary output generated by the hash function for each kmer, the data structure is updated at some level of the array using the formula $[z/2^{w+1}]$ (mod R), where z is the binary hash value and w is the number of trailing 0s in z. The value at the array level $T_w$is updated as min ($[T_w]$+1,3) [22]. At the end, the arrays are traversed to provide an estimate of the number of unique kmers by counting the number of counters with a value 1. The algorithm can be extended to other values such as the number of kmers that appear more than once and so on. KmerStream is a very efficient algorithm, as the time and space complexity of the algorithm is less than the other kmer estimation algorithms. The error rate and execution time are inversely proportional to each other.

3.1.2 Kmerlight

Kmerlight is a streaming algorithm that provides abundance histograms of all the kmers and their frequencies in DNA sequence data. Kmerlight uses the basic set up of kmerStream with a few modifications. This algorithm provides estimates of number of distinct kmers and all the kmers with frequencies ≥ 1 unlike kmerStream, which is limited to providing kmer count for unique kmers, number of distinct kmers

and total number of kmers. The algorithm consists of t instances, with each instance having M arrays, with each array containing r counters. In their implementation M = 64, which is sufficient for counting up to 2^64 kmers. Each kmer is hashed to provide a binary output (z). Depending on the number of trailing 0s, z, the level of array, w(z+1), to which the kmer goes to is calculated. Each counter of the array can hold two values <v,p> [36], where v is a counter value and p is a number from 0 to u-1 (u is a parameter). Depending on the value of z and w, the counter to which kmer goes to is selected and the value p updated every time a counter is visited. Two estimation functions are designed to calculate the number of estimated distinct kmers and kmers with different frequencies. These are calculated for each instance and a median of the estimated values is taken as the final estimate. The algorithm then generates histograms with the estimated values for distinct kmers and kmers with various counts.

3.1.3 ntCard

ntCard is also a streaming algorithm designed to provide an estimation histogram of the kmer frequencies and the number of distinct and the total number of kmers in a DNA sequence dataset. The algorithm works in three steps. In the first step, all the kmers that are passed to the algorithm as a stream are hashed using the ntHash algorithm [4]. ntHash is a rolling hash function that computes 64-bit hash values for a kmer using the hash values of the previous kmer. ntHash is proved to be one of the most efficient hashing algorithms, whose performance is substantially better than the conventional hashing algorithms [4]. The next step of the algorithm is to sample the kmers based on their 64-bit hash values. The hash values are divided into three parts, where the sample size is determined by the first part consisting s bits. Selecting the

hash values starting with s zeros, makes the sample size $1/2^s$. The third part of the hash values, called the resolution bits are used to build a multiplicity table [23] for the sample. In the third step, a statistical model is used to obtain the kmer frequency distribution from the sample distribution.

3.1.4 KmerEstimate

KmerEstimate is a streaming algorithm that provides an estimation of the count of kmers based on adaptive sampling based streaming algorithm for approximating distinct elements in a data stream [6]. Themalgorithm samples a set of kmers from the stream, by using a hash function that provides a 64-bit value. The algorithm retains kmers with rightmost s bits as all zeros for some s. For each sampled kmer, frequency is also counted. After a stream of sequences is processed, kmers of a particular frequency are counted. The algorithm is similar to ntCard, by sampling the kmers with trailing zeros greater than a certain value s in the 64-bit hash. ntCard can only have 7 and 11 as the values of s, whereas the s value is dynamic in kmerEstimate. Once the size of sample increases, s value which initially is 0, moves to 1, which doubles the size of the sample by providing a better approximation of the count and low error rate. The algorithm uses 65 hashmaps. Each time a kmer with a certain number i of trailing 0s appears in the stream, that kmer is placed in the i^{th}hashmap thus avoiding collisions, saving time and space.

3.2 Sampling Algorithms

3.2.1 KmerGenie

KmerGenie provides as an output the abundance histograms of kmers with different putative k values and also the best k value to be used in process of assembly [8]. A hash function provides a value for each kmer a hashed value, which determines whether or not the kmer is going to be a part of the sample space. Once the sample space is defined, all the kmers that belong to the sample space are then stored in a hashmap, which holds the kmer and its frequency as key and value. In the next step abundance histograms are generated for each k value and they are put through a process of fitting the model to a histogram using haploid or diploid model. In the next step, the algorithm provides the best k value for the assembly algorithms by finding the k value with maximum number of distinct kmers, which most likely covers all the genome.

The time and space complexity of the algorithm majorly depends on the factors such as read length and number of distinct kmers in a sample set. The algorithm is efficient as it provides the estimated k value using less time and memory as compared to the other estimation tools.

Chapter 4

Materials and Methods

4.1 Data Sets

Four different data sets were used to compare the performance of counting and estimation tools.

Source of this data sets is GAGE (Genome Assembly Gold-Standard Evaluations). GAGE is a University of Maryland initiative to perform an evaluation of the very latest large-scale genome assembly algorithms. Four whole-genome shotgun sequence data sets were identified for these experiments, representing a wide phylogenetic range. All data sets are illumina reads only. The genomes are:

1. *Staphylococcus aureus*
2. *Rhodobacter sphaeroides*
3. Human (chromosome 14)
4. *Bombus impatiens*

Staphylococcus aureus is a gram-positive coccal bacterium that is a member of the Firmicutes. It is frequently found in the nose, respiratory tract, and on the skin. Some information about the genome sequence is as follows:

- Average read length: 101bp

- Number of reads: 1,294,104

- Size: 0.14GB.

Rhodobacter sphaeroides is a kind of purple bacteria; a group of bacteria that can obtain energy through photosynthesis. Most notably, Rhodobacter sphaeroides is closely studied as a model organism for an oxygenic photosynthesis and carbonization. Some information about the genome sequence is as follows:

- Average read length: 101bp

- Number of reads: 2,050,868

- Size: 0.22GB.

Human Chromosome 14 is one of the 23 pairs of chromosomes in humans. This chromosome spans about 107 million base pairs and represents between 3 and 3.5 percent of the total DNA in the cells. Some information about the genome sequence is as follows:

- Average read length: 101bp

- Number of reads: 36,504,800

- Size: 4GB.

Bombus impatiens known as eastern bumble bee is the commonly seen bumble bee in most of eastern North America. They are known as one of the most important species of pollinator bees in North America [33]. Some information about the genome data is as follows:

- Avg read length: 124bp

- Number of reads: 303,118,594

- Size: 48GB.

4.2 Computing Environment

Computing environment specifications are as follows.

RAM: 30GB

VCPUs: 4

Disk Space: 160GB.

All the tools were executed in the same environment with the four data sets mentioned above. Their total RAM usage and run time is compared and their performance is discussed in the later sections.

KAnalyze, Khmer, Jellyfish, Squeakr and GeneomeTester4 can perform kmer counting upto k = 32. For a fair comparison, all the tools are tested for k = 21 and 31 and we conducted our experiments with out parallelization, i.e, number of threads for execution is always 1.

Kmer estimation tools were tested for k = 21,31,41,51 and 61. For the purpose of this research, KmerStream and KmerGenie are executed with a single thread, while KmerEstimate, ntCard and Kmerlight are executed with default number of threads as they do not provide an option to the user to select the number of threads.

Chapter 5

Results and Discussion

5.1 Kmer Counting tools

5.1.1 Execution Time Comparison

Figures 5.1~5.5 show comparison of the performance of 13 kmer counting tools.

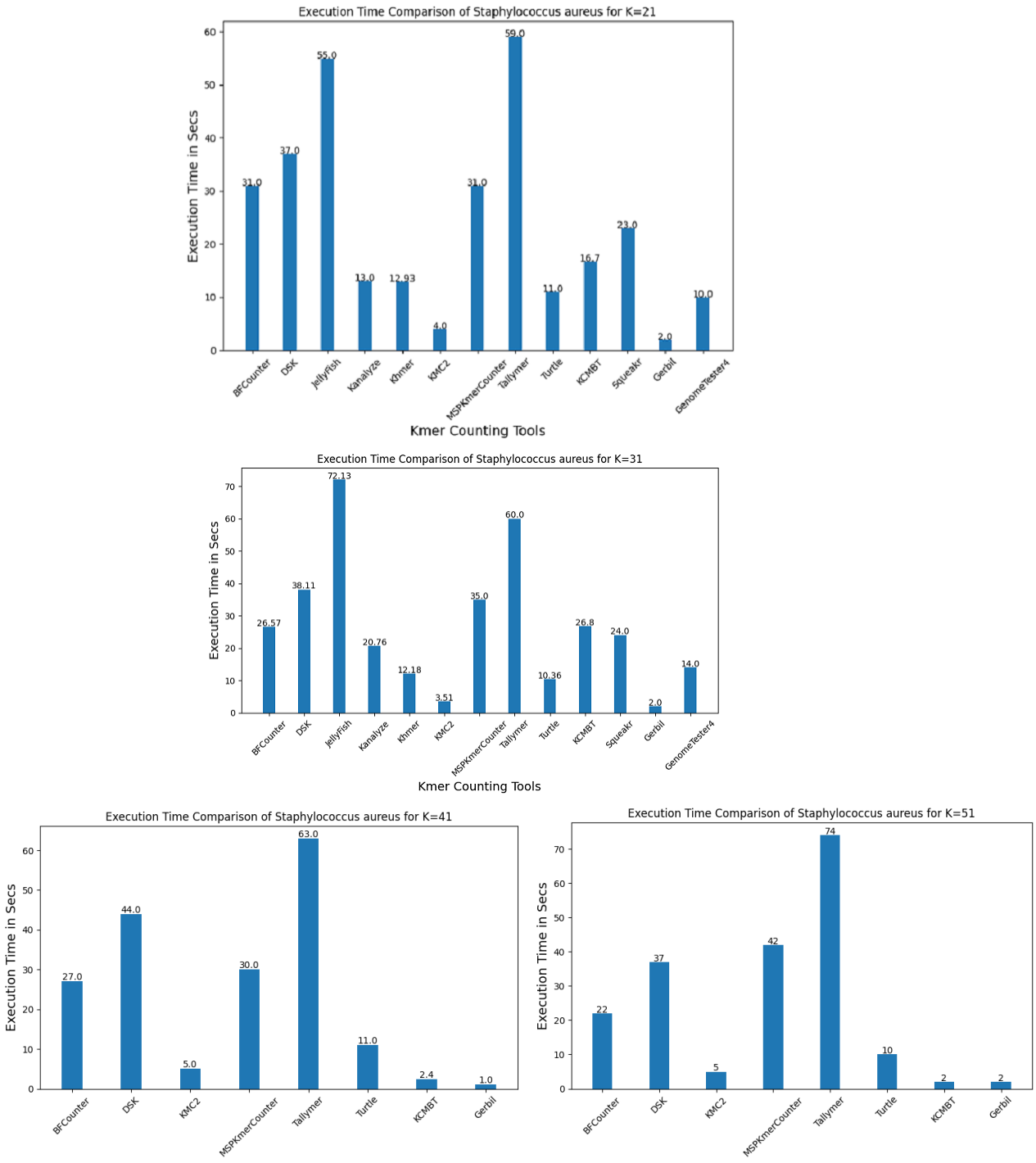

Figure 5.1: Execution Time Comparison of Kmer Counting Tools - *Staphylococcus aureus*

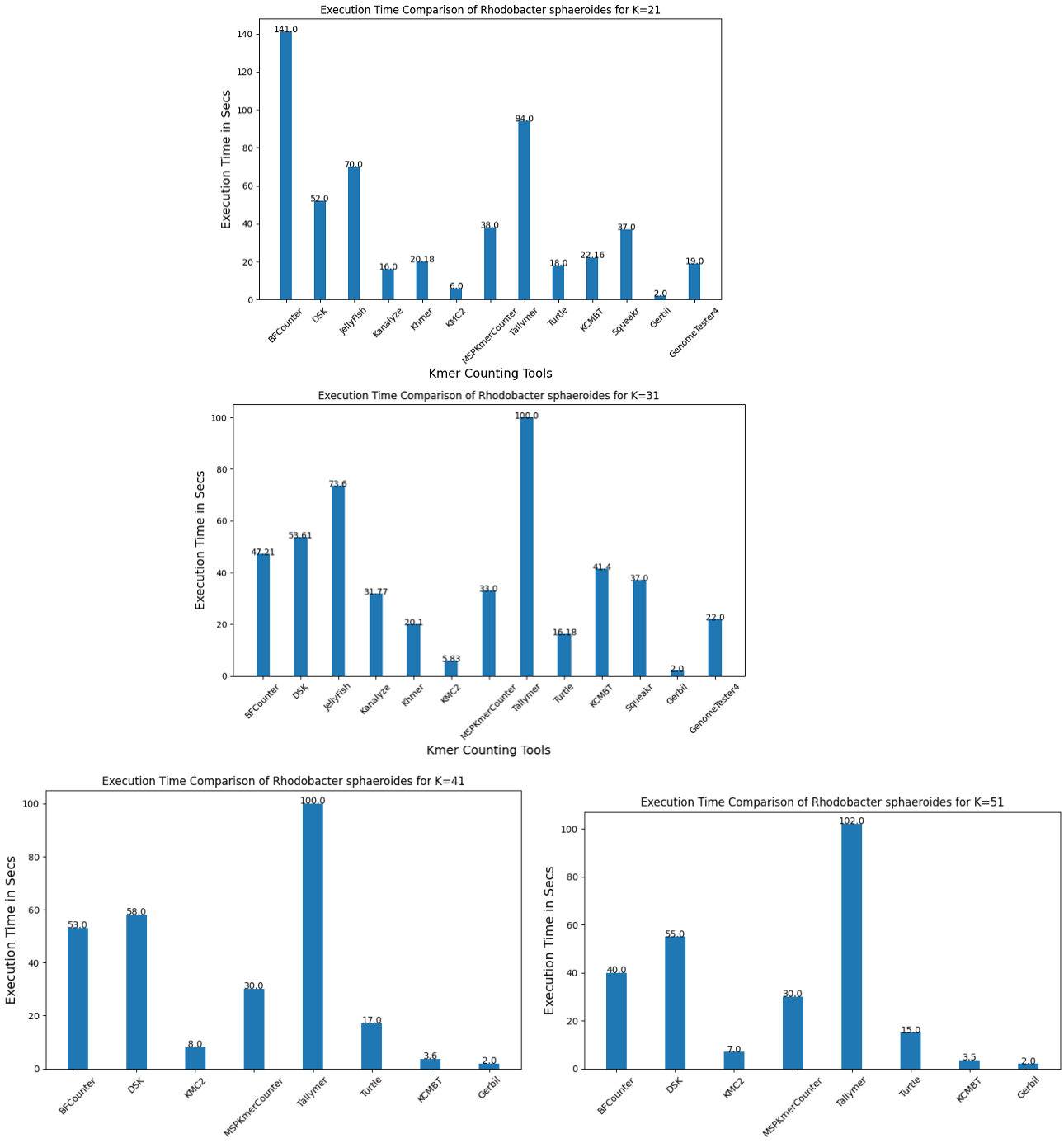

Figure 5.2: Execution Time Comparison of Kmer Counting Tools - *Rhodobacter sphaeroides*

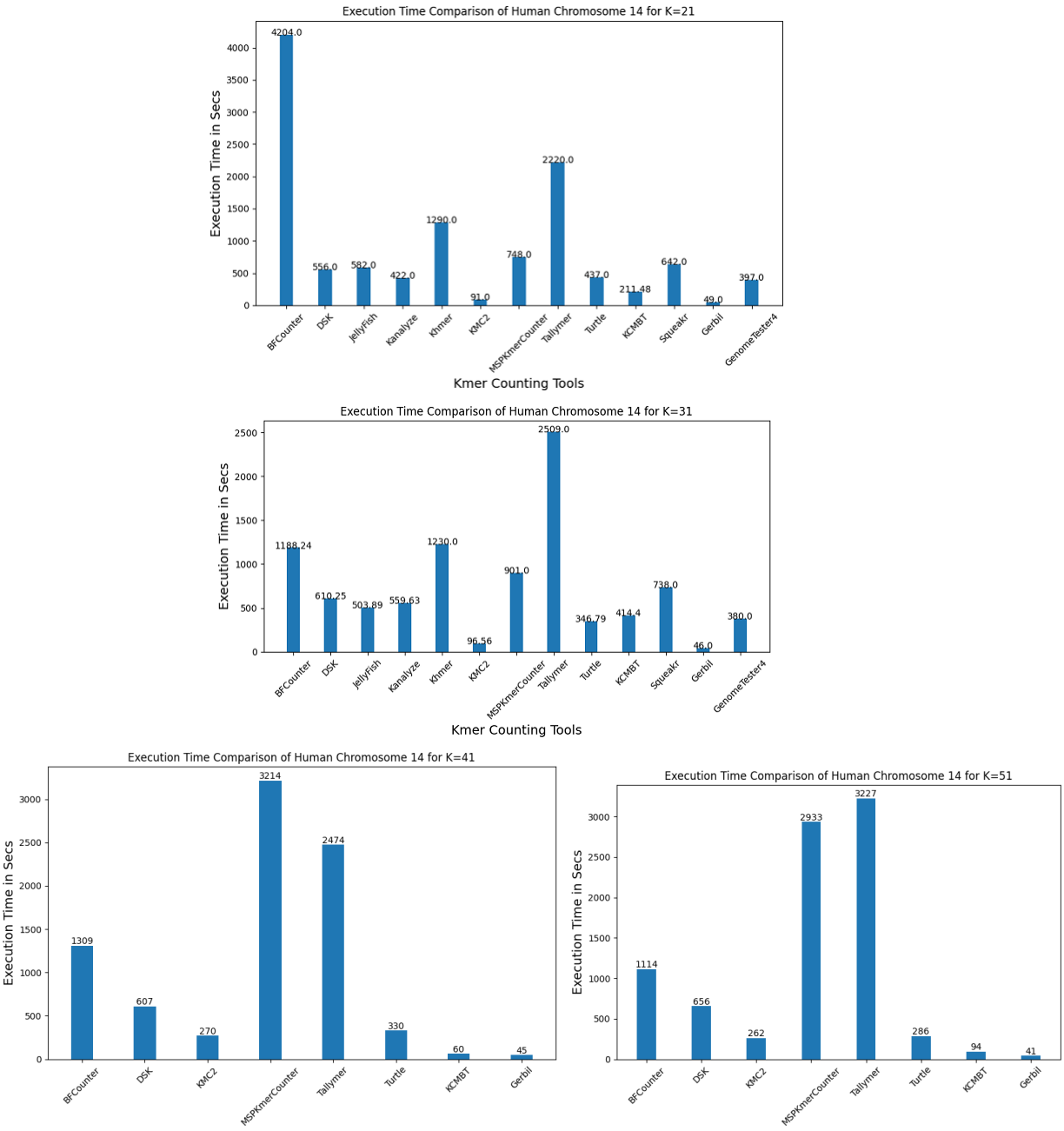

Figure 5.3: Execution Time Comparison of Kmer Counting Tools - Human Chromosome 14

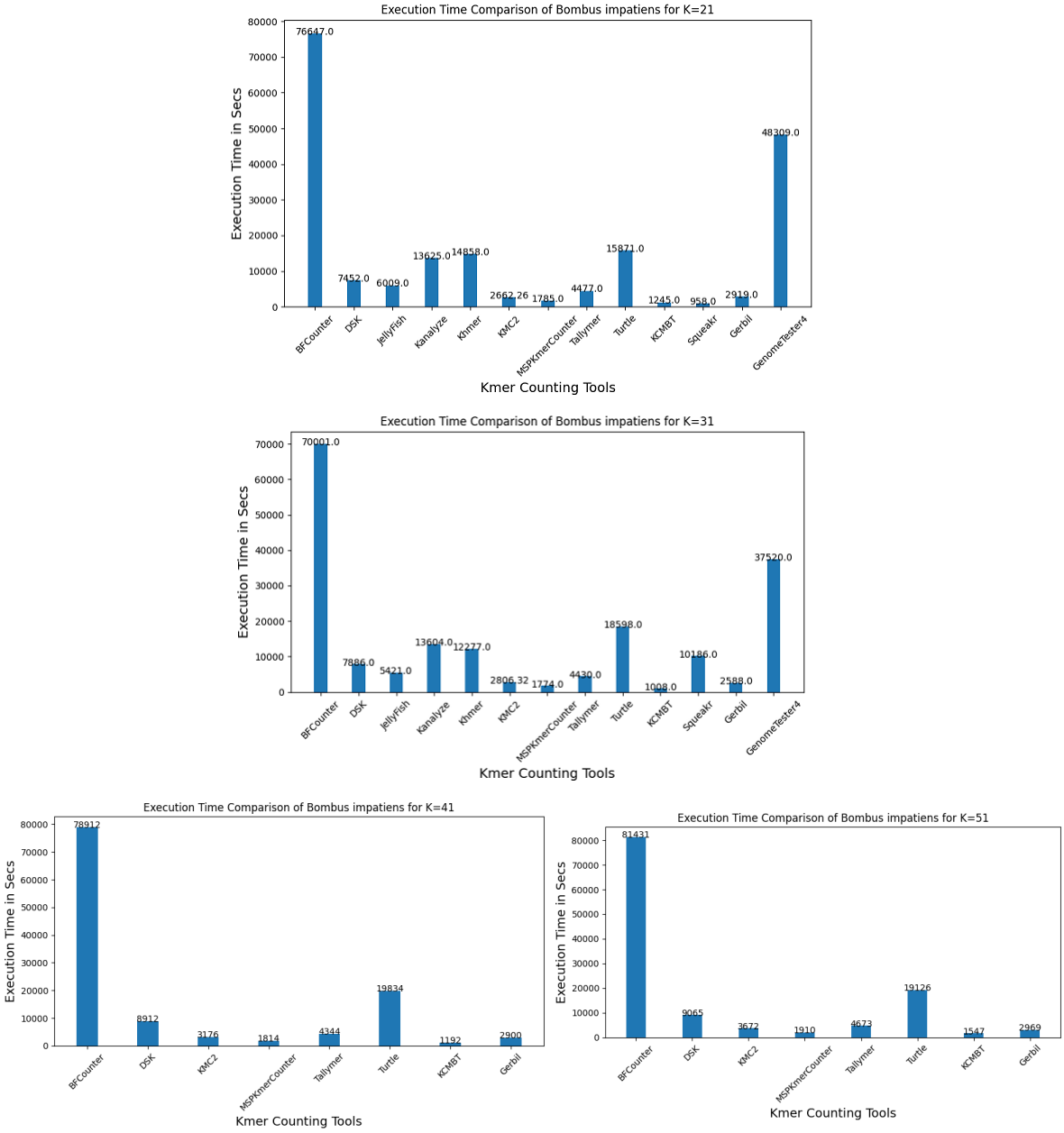

Figure 5.4: Execution Time Comparison of Kmer Counting Tools - *Bombus impatiens*

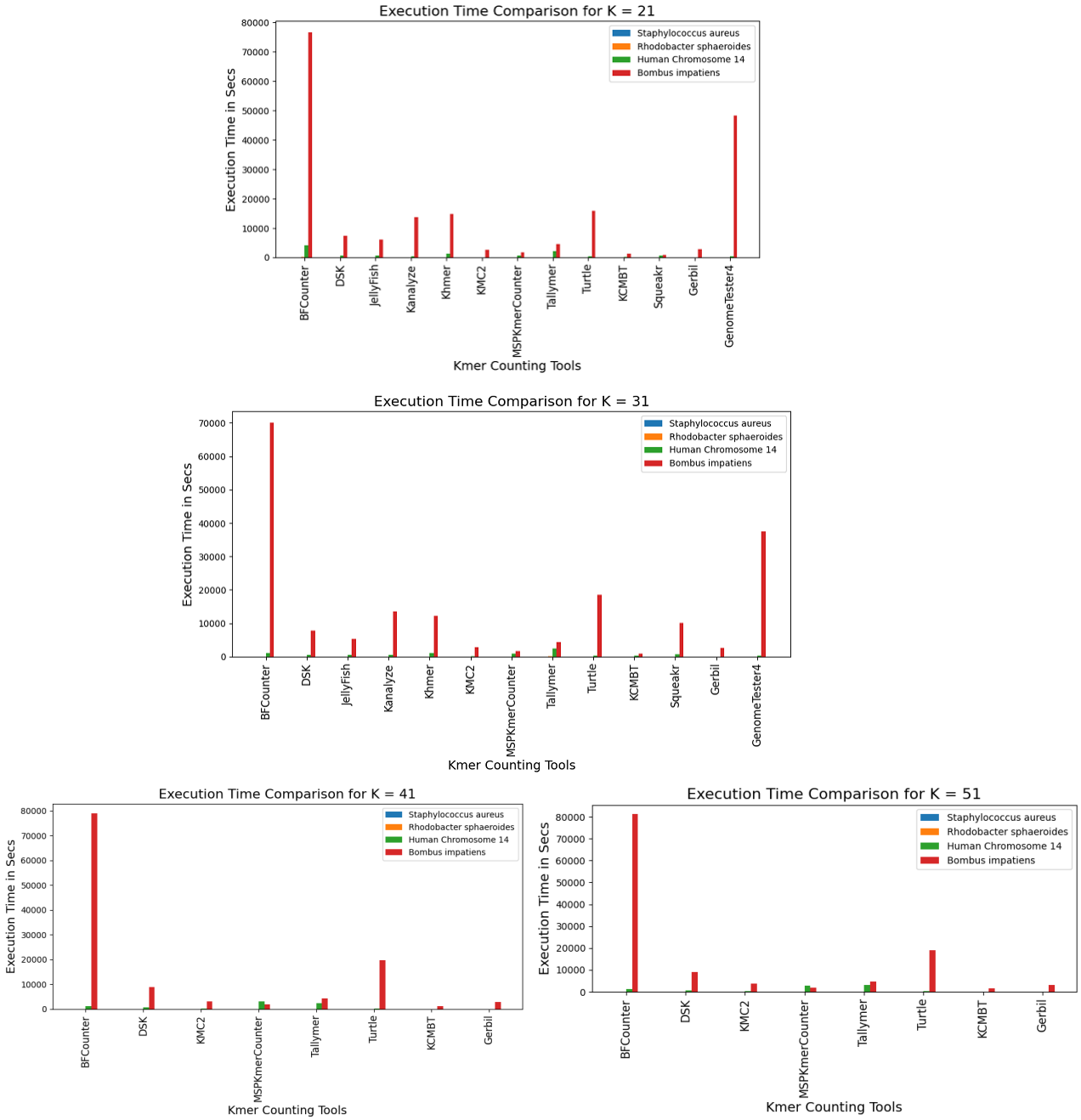

Figure 5.5: Execution Time Comparison of Kmer Counting Tools

We observed that KMC2, Khmer and Gerbil are the most efficient algorithms with least execution time, followed by KCMBT and Turtle. While DSK is almost consistent with its execution time for two different k values, the execution time varies

for GenomeTester4 and MSPKmerCounter depending on the k values and dataset. Finally BFCounter and Tallymer's execution time is the highest as compared to the others for the k values we tested.

5.1.2 Memory Usage Comparison

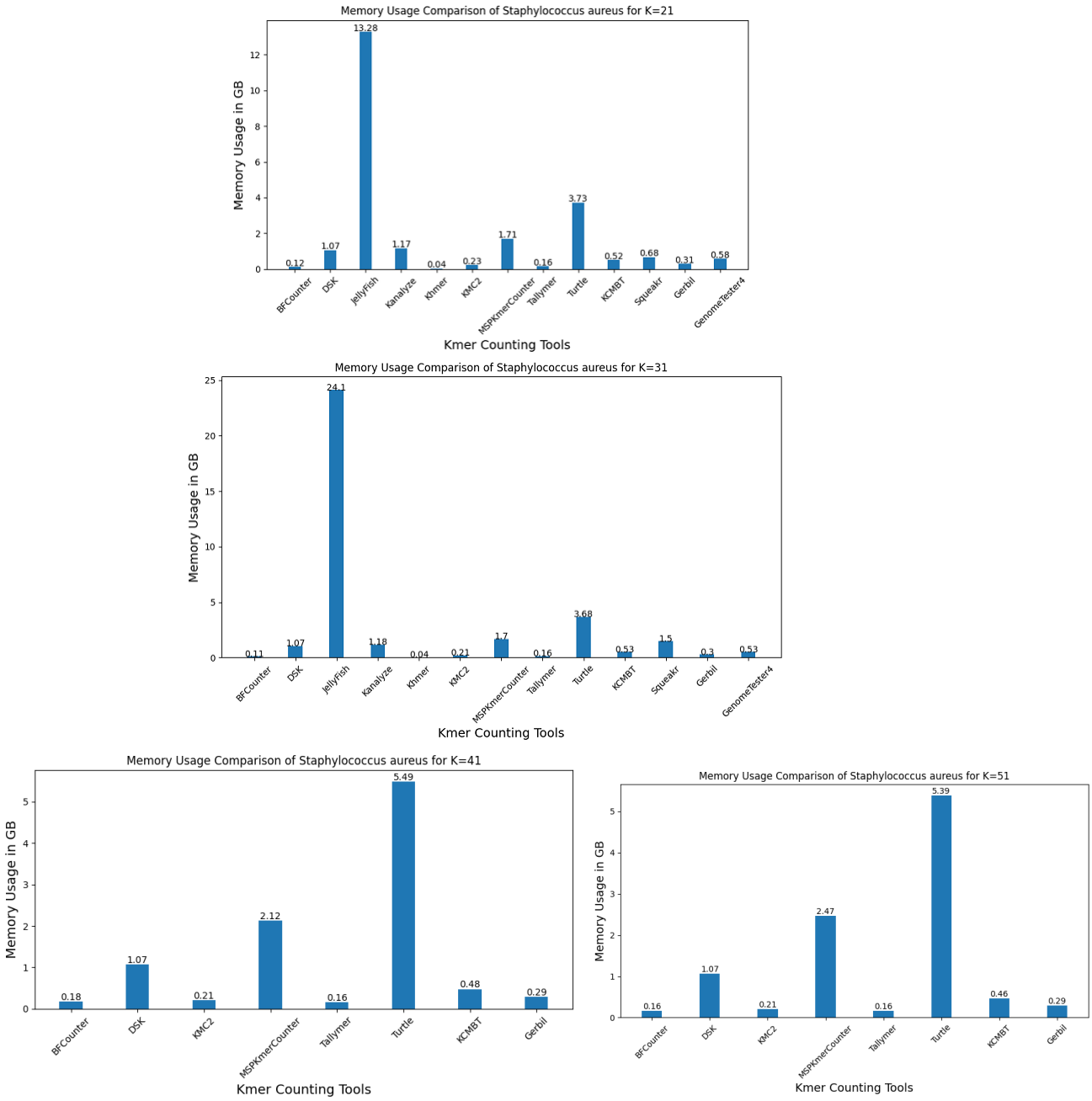

Figure 5.6: Memory Usage Comparison of Kmer Counting Tools - *Staphylococcus aureus*

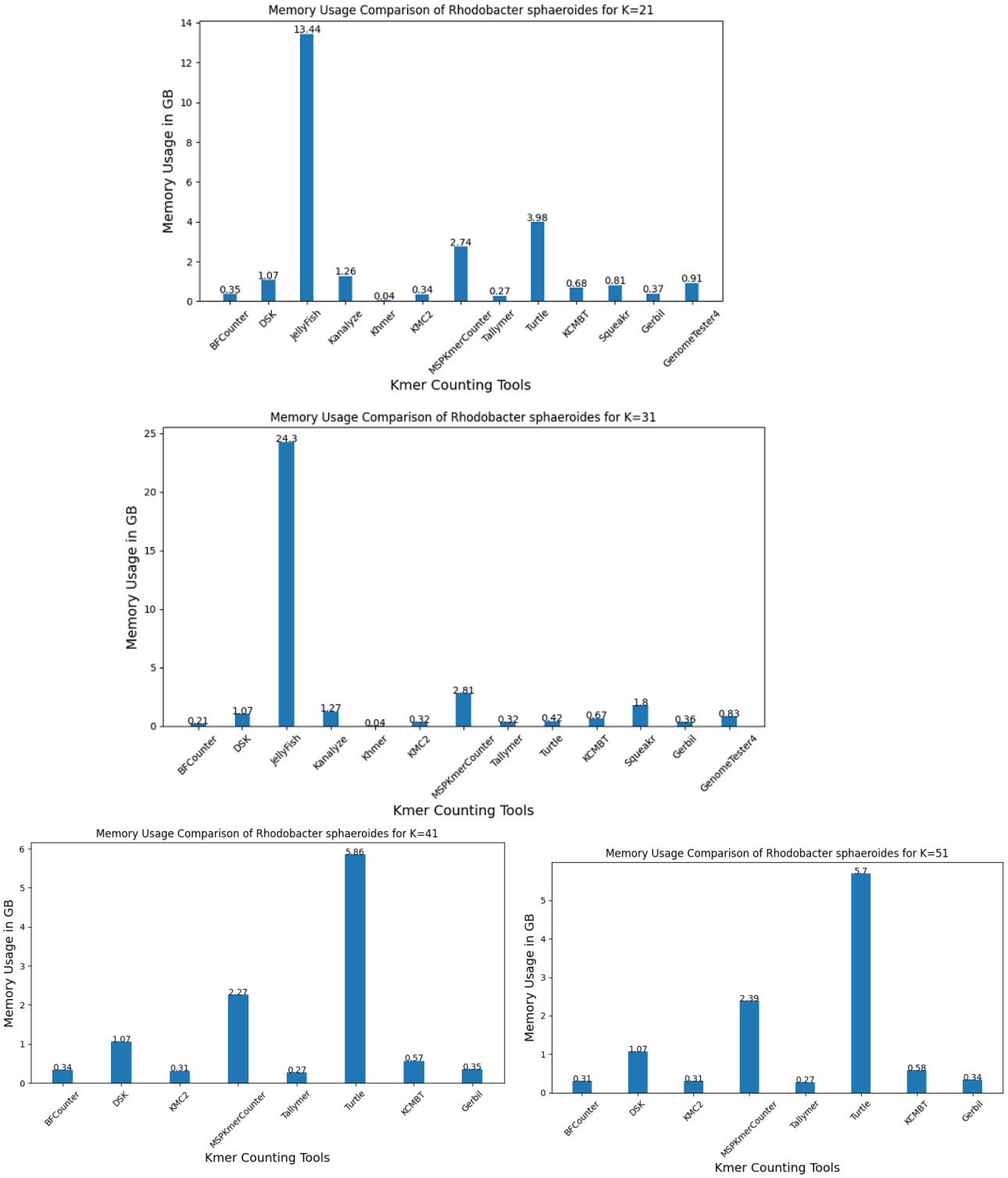

Figure 5.7: Memory Usage Comparison of Kmer Counting Tools - *Rhodobacter sphaeroides*

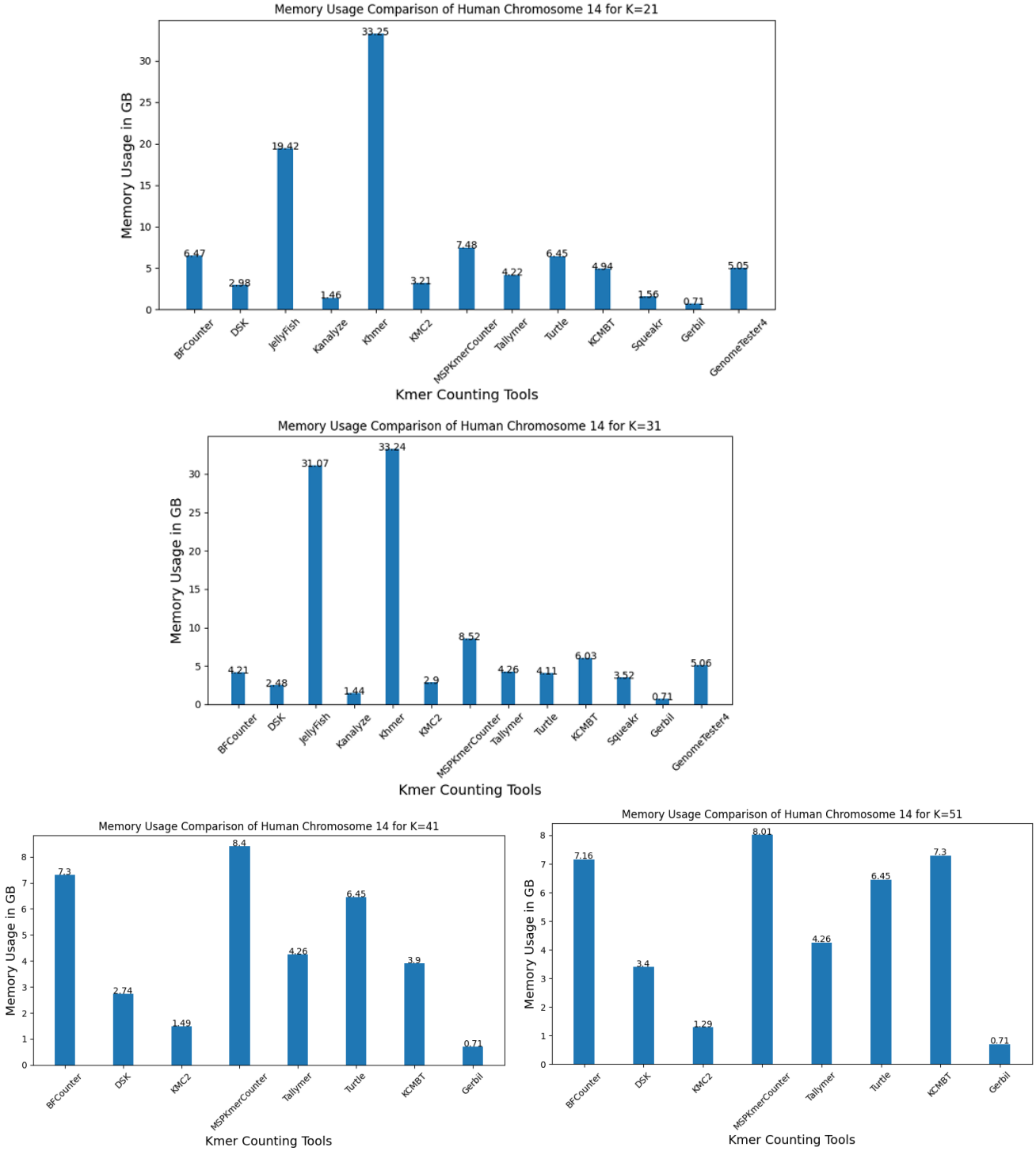

Figure 5.8: Memory Usage Comparison of Kmer Counting Tools - Human Chromosome 14

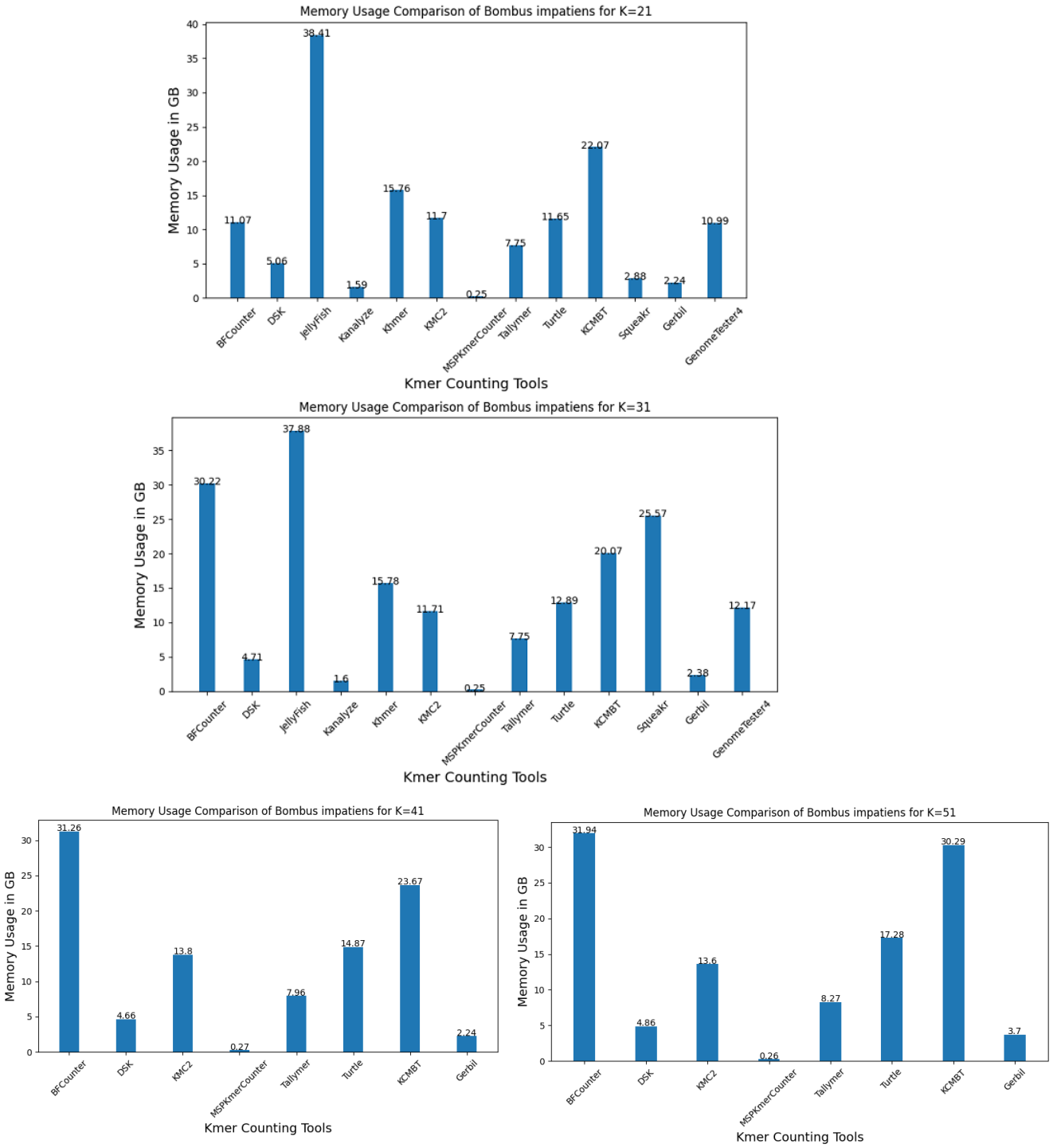

Figure 5.9: Memory Usage Comparison of Kmer Counting Tools - *Bombus impatiens*

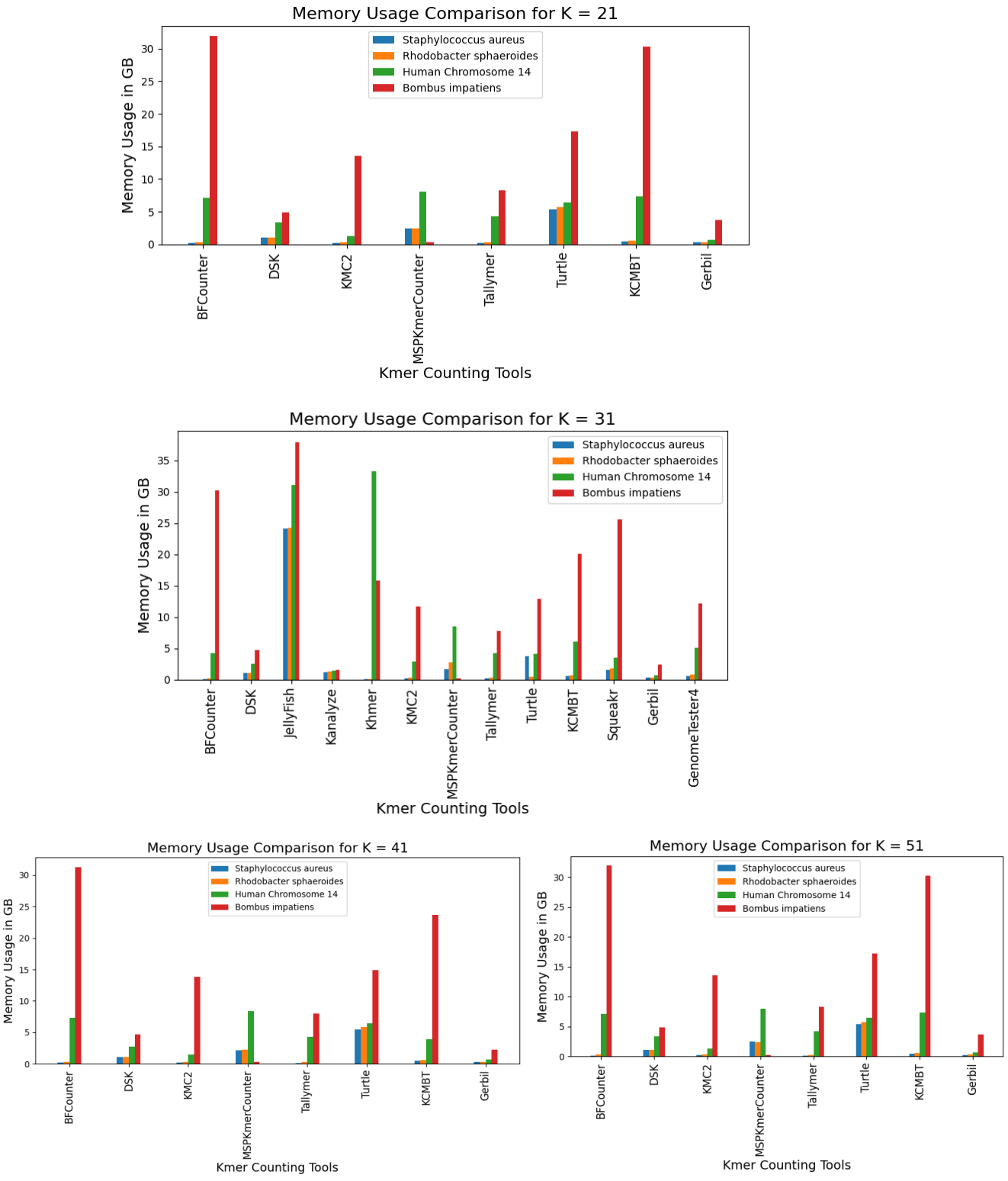

Figure 5.10: Memory Usage Comparison of Kmer Counting Tools

Figures 5.10 show comparison of RAM utilization among 13 kmer counting methods with different k values. Clearly, KMC2, Kanalyze and Gerbil are the most efficient algorithms with least RAM utilization followed by DSK, Turtle and KCMBT. Jellyfish and BFCounter and have the most RAM utilization for different k values.

5.1.3 Discussion - Kmer Counting Tools

This section provides a performance analysis of kmer counting tools.

5.1.3.1 Bloom Filter & Count Min Sketch Algorithms:

BFCounter and Turtle are bloom-filter based and Khmer is count-min sketch based algorithms. Run time of these tools is comparable to the other kmer counting tools. The memory requirement of these tools increases as the size of the data set increases. Since bloom filter uses multiple hash tables to reduce collisions, as the number of kmers increases, the size and number of the hash tables grows to avoid collisions. When the maximum memory that they can use was restricted to a certain amount, their execution time increases. Figures 5.3 (Human chromosome 14 size 4GB), 5.4 (*Bombus impatiens* size 48GB) show that as the maximum RAM that can be used was capped at 30GB, execution time of BFCounter increased significantly from 1188.24secs to 70001secs for k=31 as the size of datasets increased from 4GB to 48GB.

5.1.3.2 Disk Partitioning Algorithms:

Disk partitioning tools that are being accessed are DSK, KMC2, MSPKmerCounter and KAnalyze. These are the ones with the most efficient run time and RAM usage. From Figures 5.5 and 5.10, it is evident that KMC is the one with least run time for the one of the larger dataset Human Chromosome 14 and KAnalyze is the tool with

least RAM utilization as compared to the other kmer counting tools, which can be attributed to minimizers with partitioning technique.

5.1.3.3 LCP Algorithms:

Tallymer uses longest common prefix technique. Figure 5.8 shows that while Tallymer uses a moderate amount of RAM for the largest dataset of size 7.75 GB, it has the highest run time for the same data set as shown in figure 5.4. Tallymer involves sorting of enhanced suffix arrays and construction of lcp tree and the major portion of run time goes into sorting.

5.1.3.4 Lock Free Queues Algorithms:

Jellyfish is the tool that uses lock free queues. Jellyfish with single thread is seen to perform Ill with moderate execution time from figure 5.5, but it is very memory intensive from figure 5.10. It used the maximum memory as compared to the other tools.

5.1.3.5 Multiple Burst Tree Algorithms:

KCMBT is a relatively new tool, which shows a good performance with both run time from figure 5.5 and RAM usage from figure 5.10. Although it is not the fastest as compared to the other kmer counting tools, considering the run time and RAM usage together it is one of the most efficient counting tools. This can be attributed to the data structure burst tress, which are faster than the binary search trees and utilize less memory.

5.1.3.6 Minimizers:

KMC2 and MSPKmerCounter both use the concept of minimizers where a set of m-mers (m<k) are formed and then overlap between these m-mers provides the network of kmers. The ability to perform parallel processing and automatic parameter selection also provides a lot of flexibility and plays an important role in the better performance of these tools.

5.1.3.7 Counting Quotient Filter Based:

Squeakr is the tool that used CQF approach. Squeakr version that provides an approximate number of kmer and not the exact number was used for the purpose of this research. The CQF approach coupled with murmurhash function and multi-threading makes the performance of this tool better than some of the kmer counting tools. The only difference between exact and approximate is that approximate squeakr version allows a small false positive rate.

5.1.3.8 GPU Computing:

Gerbil stands out as one of the better performing counting tools. This can be attributed to the two-disk approach that Gerbil implements and the load balancing between GPU and CPU.

5.1.3.9 SAC:

GenomeTester4 is the tool that used sorting and counting approach. This is one of the simple approaches that utilizes the memory efficiently. The temporary usage of arrays to store the reads and multi-threading approach to sort and count the kmers makes this tool memory efficient.

5.2 Kmer Estimation Tools

There are 5 kmer estimation tools that are being analyzed as a part of this research. The following tables and figures provide comparisons of execution time of these tools.

5.2.1 Execution Time Comparison

Table 5.1: Execution Time Comparison for K=21

Data Set & K=21	KmerEstimate	KmerGenie	KmerStream	Kmerlight	ntCard
Staphylococcus aureus	20	40	6	4.87	3
Rhodobacter sphaeroides	28	44	3	7.66	4
Human Chromosome 14	111	561	39	244	25
Bombus impatiens	1841	5629	1521	961	1764

Table 5.2: Execution Time Comparison for K=31

Data Set & K=31	KmerEstimate	KmerGenie	KmerStream	Kmerlight	ntCard
Staphylococcus aureus	18	34.67	2.57	4.4	4
Rhodobacter sphaeroides	25	43.13	3.22	6.63	5
Human Chromosome 14	110	530.67	37.15	405.32	23
Bombus impatiens	1840	5337	2047	609	2075

Table 5.3: Execution Time Comparison for K=41

Data Set & K=41	KmerEstimate	KmerGenie	KmerStream	Kmerlight	ntCard
Staphylococcus aureus	17	11	2	3.21	4
Rhodobacter sphaeroides	24	15	3	5.06	5
Human Chromosome 14	98	434	33	81	21
Bombus impatiens	1841	5629	1521	961	1764

Table 5.4: Execution Time Comparison for K=51

Data Set & K=51	KmerEstimate	KmerGenie	KmerStream	Kmerlight	ntCard
Staphylococcus aureus	15	11	2	3.21	3
Rhodobacter sphaeroides	21	15	3	5.06	4
Human Chromosome 14	95	434	33	81	20
Bombus impatiens	1837	4664	1778	1769	1765

Table 5.5: Execution Time Comparison for K=61

Data Set & K=61	KmerEstimate	KmerGenie	KmerStream	Kmerlight	ntCard
Staphylococcus aureus	16	11	2	3.21	4
Rhodobacter sphaeroides	25	15	3	5.06	5
Human Chromosome 14	224	434	33	81	22
Bombus impatiens	1835	4410	1778	1770	2059

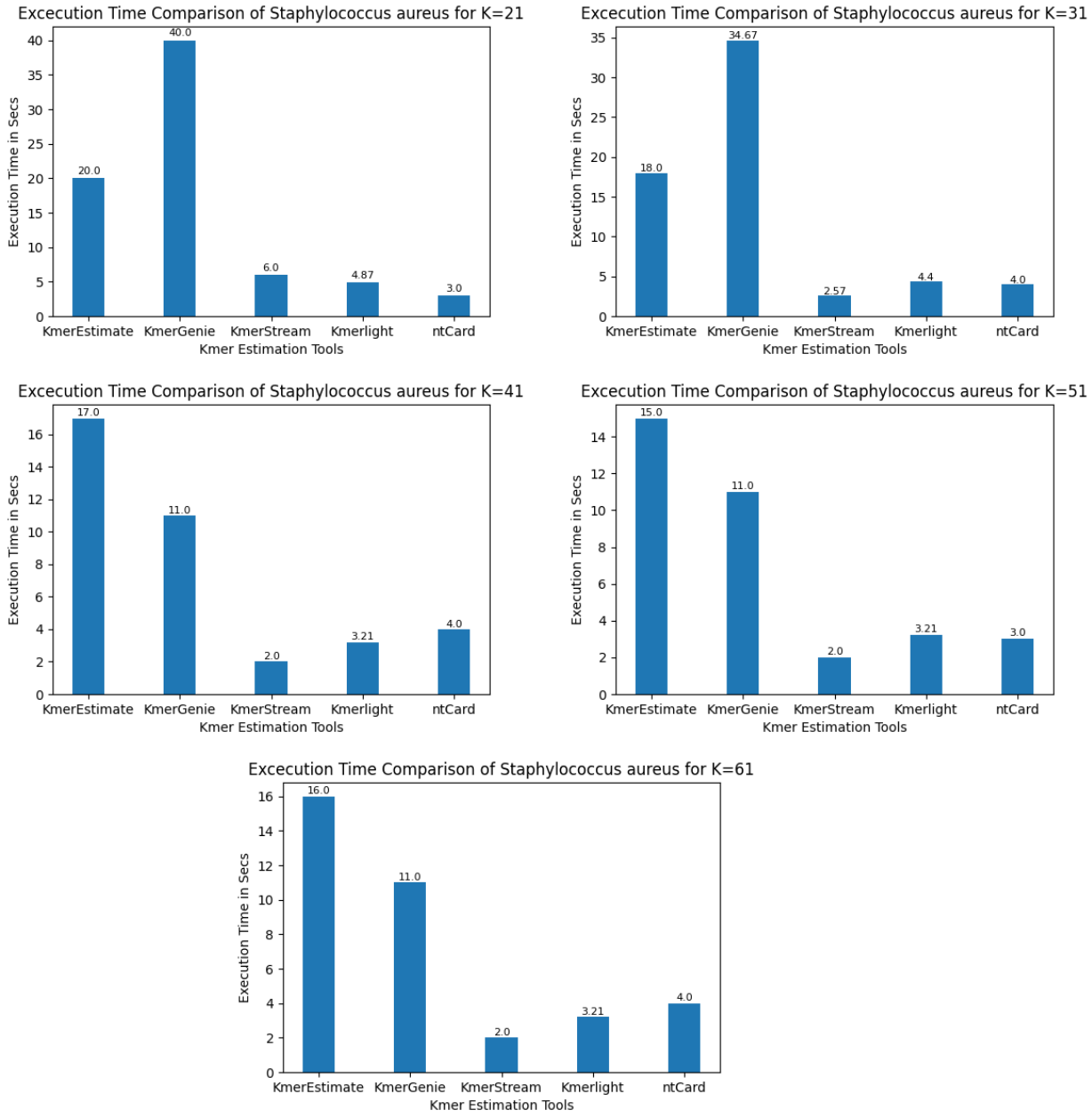

Figure 5.1: Execution Time Comparison of Kmer Estimation Tools for *Staphylococcs aureus*

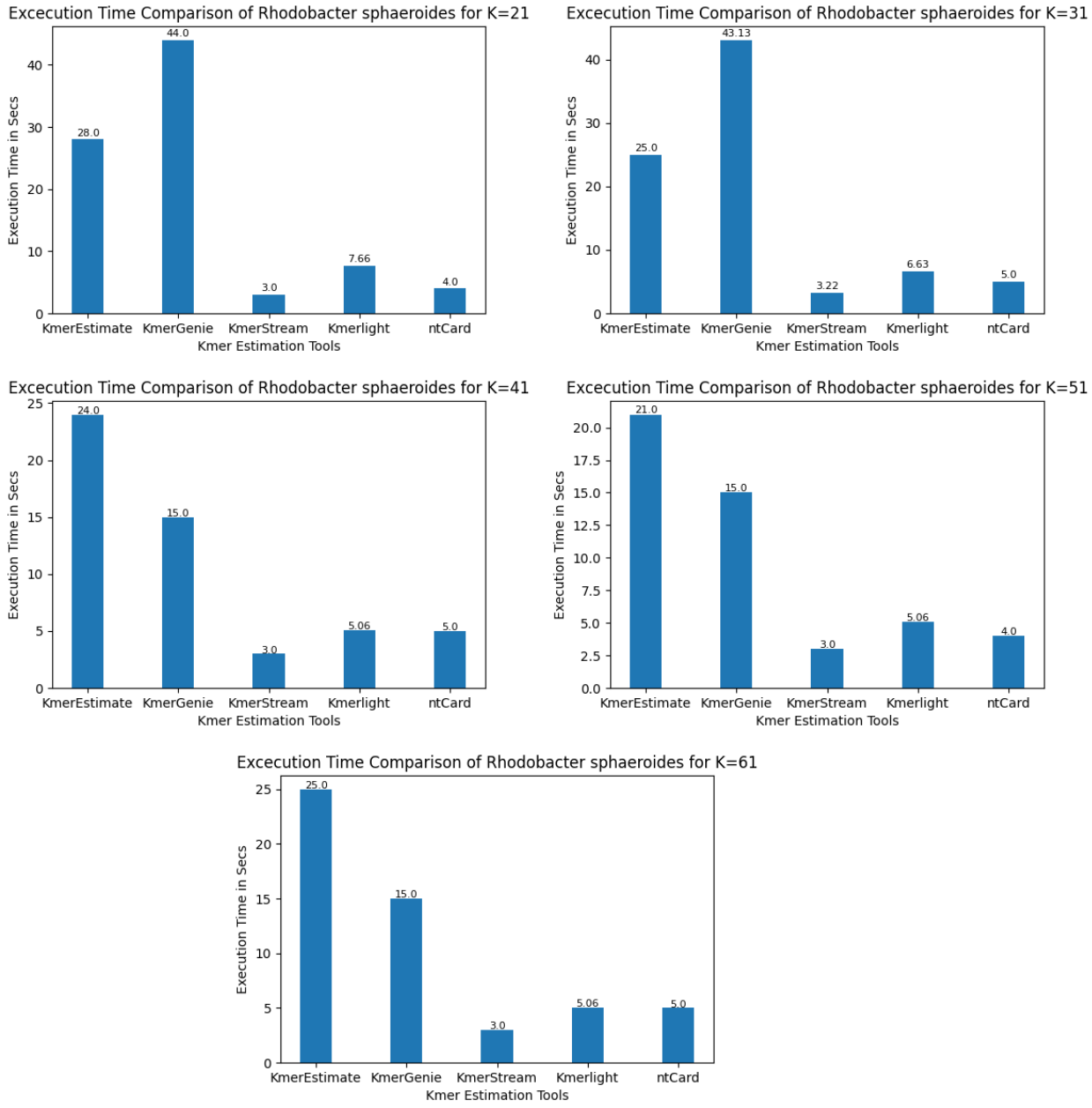

Figure 5.2: Execution Time Comparison of Kmer Estimation Tools for *Rhodobacter sphaeroides*

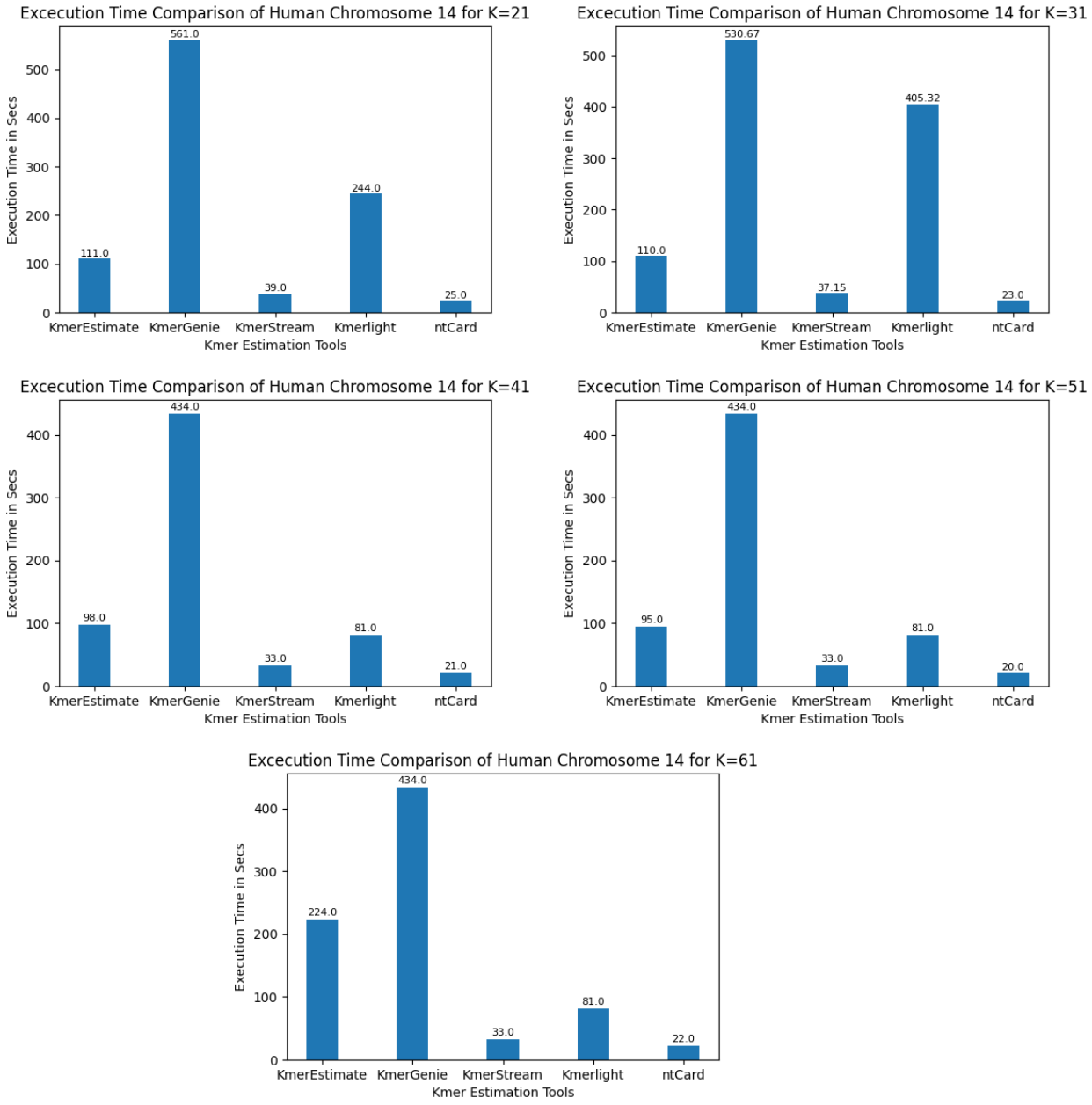

Figure 5.3: Execution Time Comparison of Kmer Estimation Tools for the Human chromosome 14

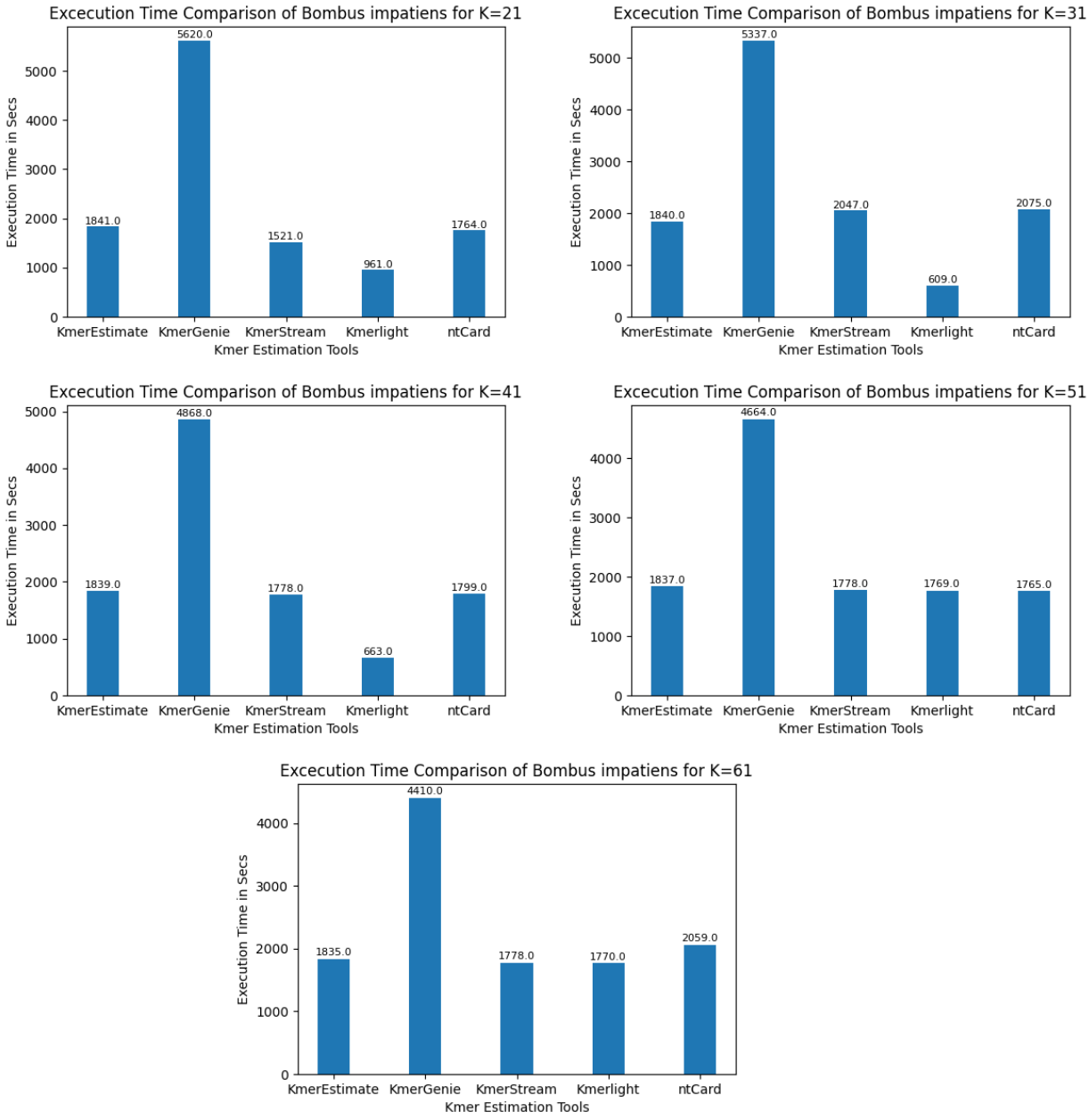

Figure 5.4: Execution Time Comparison of Kmer Estimation Tools for *Bombus impatiens*

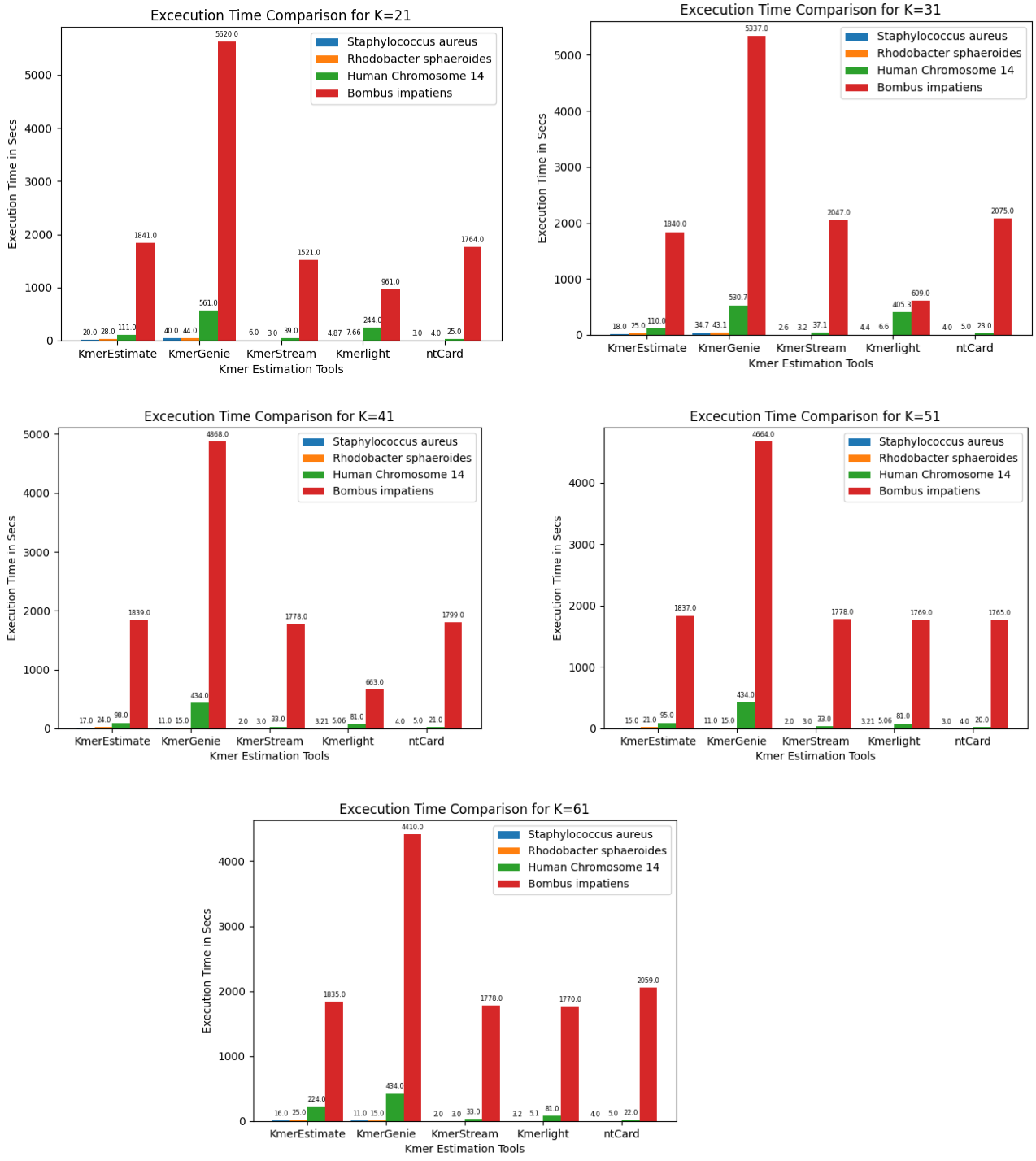

Figure 5.5: Execution Time Comparison of Kmer Estimation Tools for different K-values

Figure 5.5 shows that KmerGenie has the highest run time followed by kmerlight, kmerStream, kmerEstimate and ntCard.

5.2.2 Memory Usage Comparison

The following tables and figures provide comparisons of the memory usage of kmer estimation tools for k = 21,31,41,51 and 61.

Table 5.6: Memory Usage Comparison for K=21

Data Set & K=21	KmerEstimate	KmerGenie	KmerStream	Kmerlight	ntCard
Staphylococcus aureus	0.45	0.15	0.01	0.40	0.50
Rhodobacter sphaeroides	0.51	0.16	0.01	0.40	0.50
Human Chromosome 14	0.54	0.15	0.01	0.45	0.50
Bombus impatiens	0.54	0.13	0.01	0.26	0.50

Table 5.7: Memory Usage Comparison for K=31

Data Set & K=31	KmerEstimate	KmerGenie	KmerStream	Kmerlight	ntCard
Staphylococcus aureus	0.45	0.15	0.01	0.40	0.50
Rhodobacter sphaeroides	0.51	0.16	0.01	0.39	0.50
Human Chromosome 14	0.54	0.16	0.01	0.44	0.50
Bombus impatiens	0.56	0.13	0.01	0.28	0.50

Table 5.8: Memory Usage Comparison for K=41

Data Set & K=41	KmerEstimate	KmerGenie	KmerStream	Kmerlight	ntCard
Staphylococcus aureus	0.43	0.14	0.01	0.38	0.50
Rhodobacter sphaeroides	0.51	0.16	0.01	0.38	0.50
Human Chromosome 14	0.54	0.15	0.01	0.44	0.50
Bombus impatiens	0.54	0.13	0.01	0.26	0.50

Table 5.9: Memory Usage Comparison for K=51

Data Set & K=51	KmerEstimate	KmerGenie	KmerStream	Kmerlight	ntCard
Staphylococcus aureus	0.40	0.14	0.01	0.38	0.50
Rhodobacter sphaeroides	0.51	0.16	0.01	0.38	0.50
Human Chromosome 14	0.53	0.15	0.01	0.44	0.50
Bombus impatiens	0.55	0.13	0.01	0.25	0.50

Table 5.10: Memory Usage Comparison for K=61

Data Set & K=61	KmerEstimate	KmerGenie	KmerStream	Kmerlight	ntCard
Staphylococcus aureus	0.30	0.14	0.01	0.38	0.50
Rhodobacter sphaeroides	0.50	0.16	0.01	0.38	0.50
Human Chromosome 14	0.52	0.15	0.01	0.44	0.50
Bombus impatiens	0.54	0.13	0.01	0.26	0.50

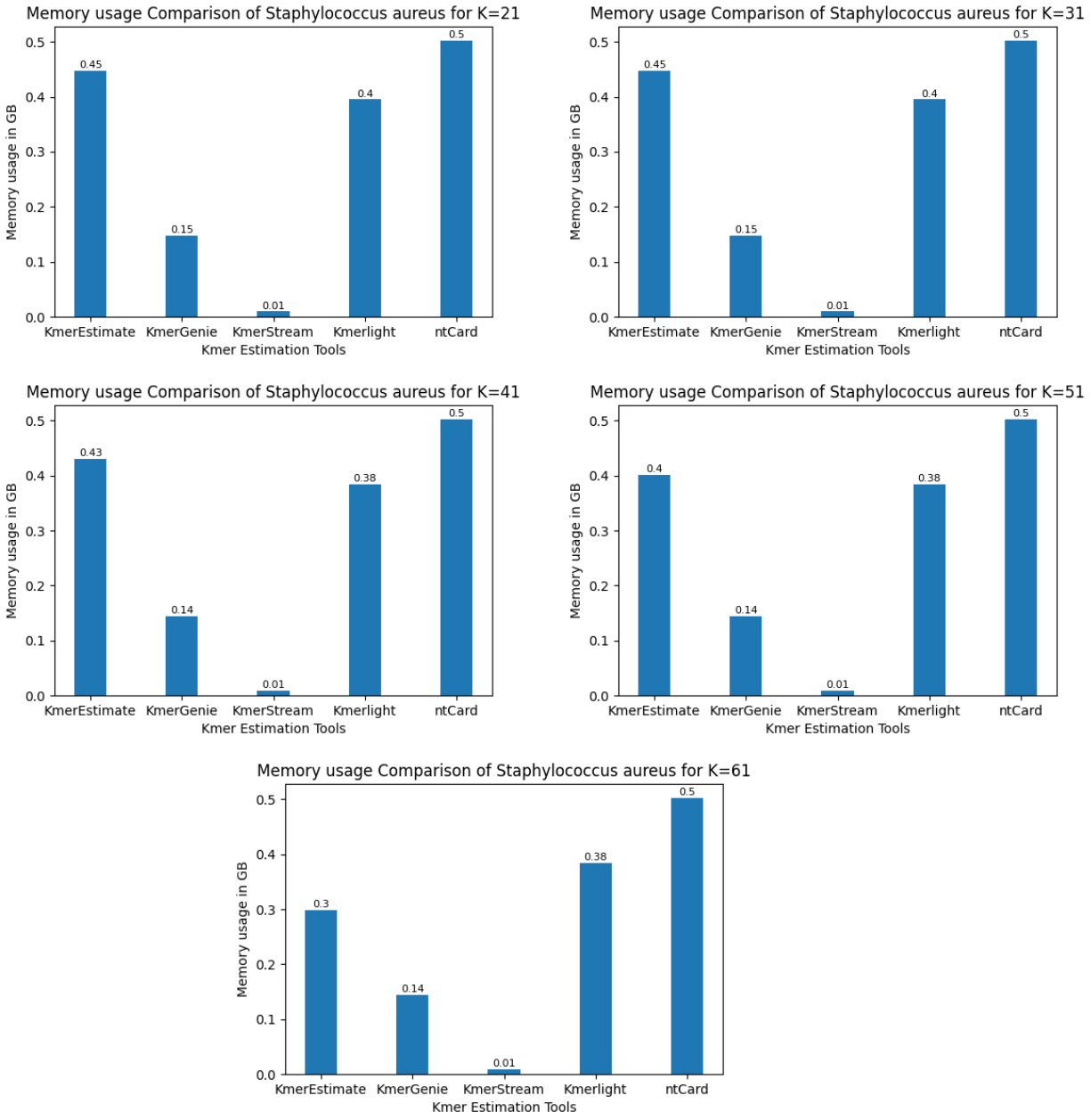

Figure 5.6: Memory Usage Comparison of Kmer Estimation Tools for *Staphylococcus aureus*

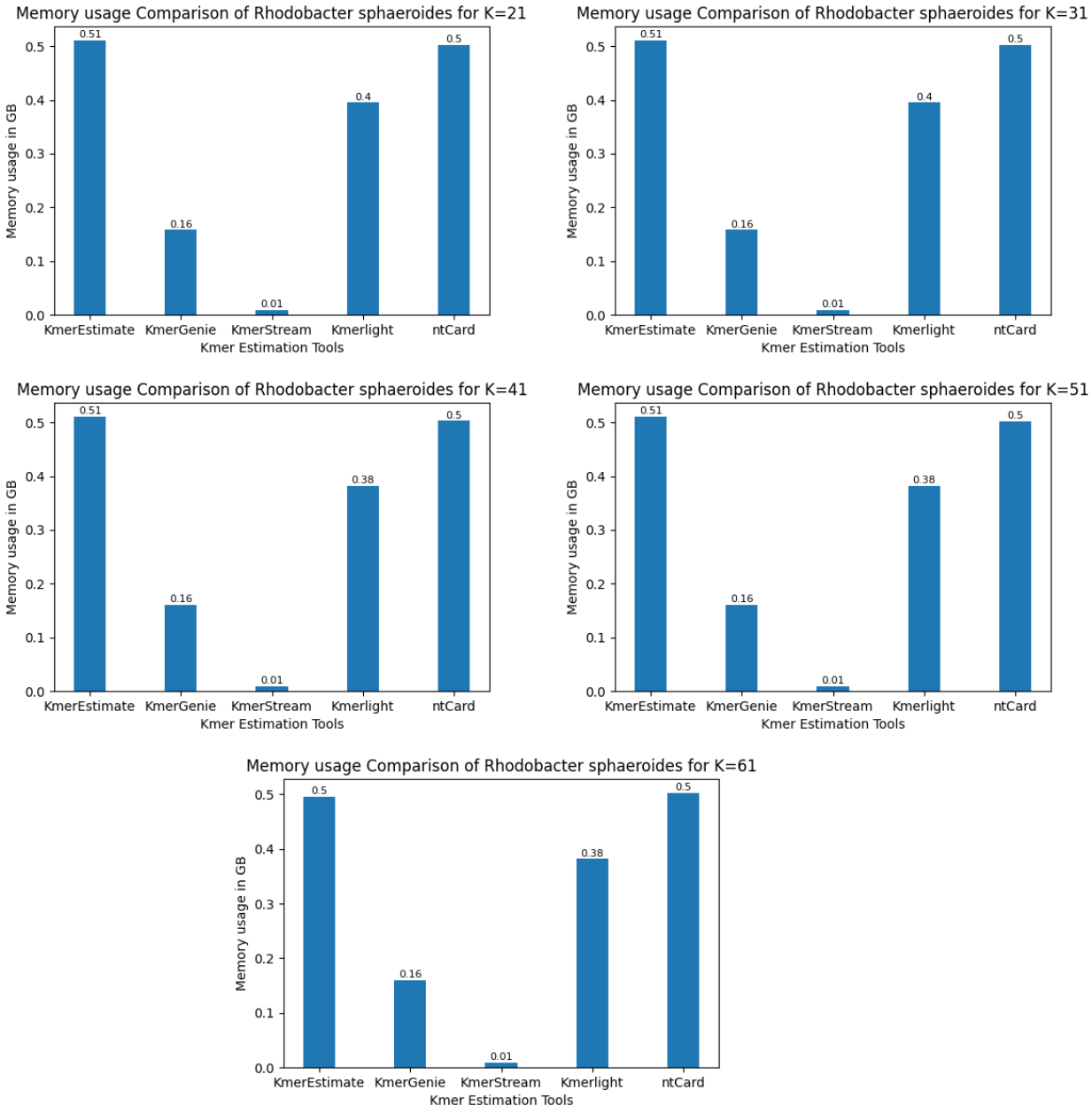

Figure 5.7: Memory Usage Comparison of Kmer Estimation Tools for *Rhodobacter sphaeroides*

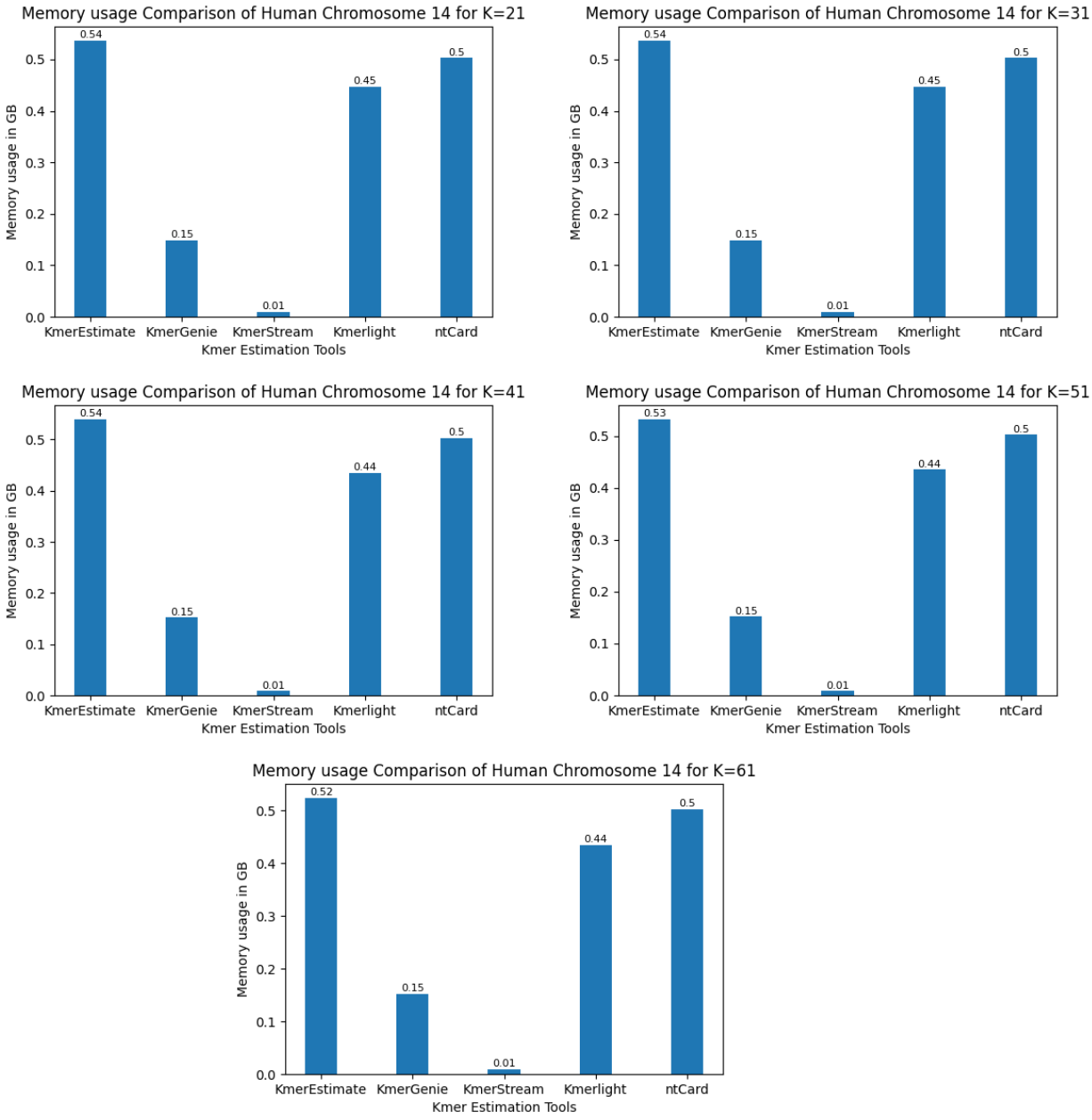

Figure 5.8: Memory Usage Comparison of Kmer Estimation Tools for the Human Chromosome 14

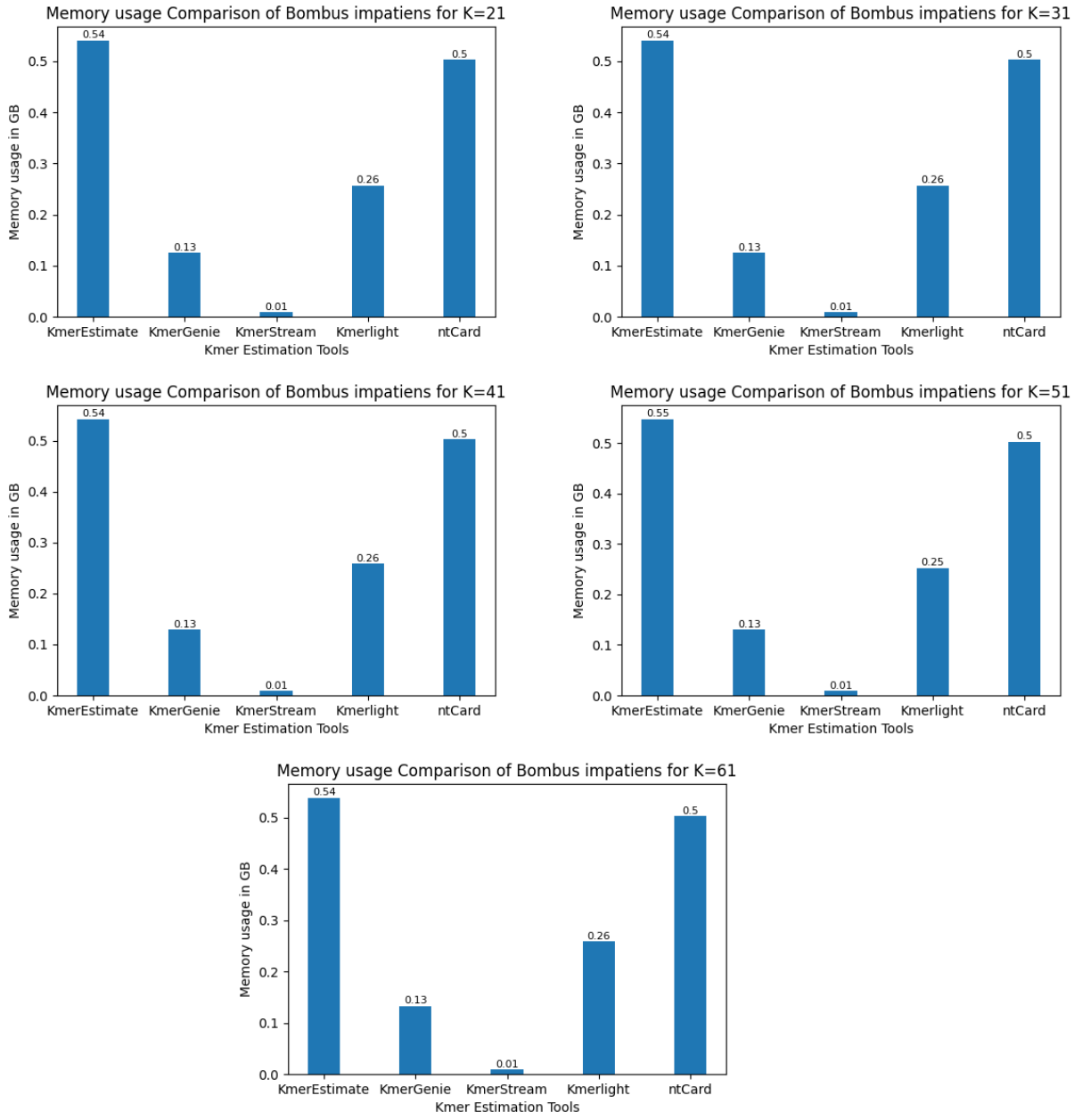

Figure 5.9: Memory Usage Comparison of Kmer Estimation Tools for *Bombus impatiens*

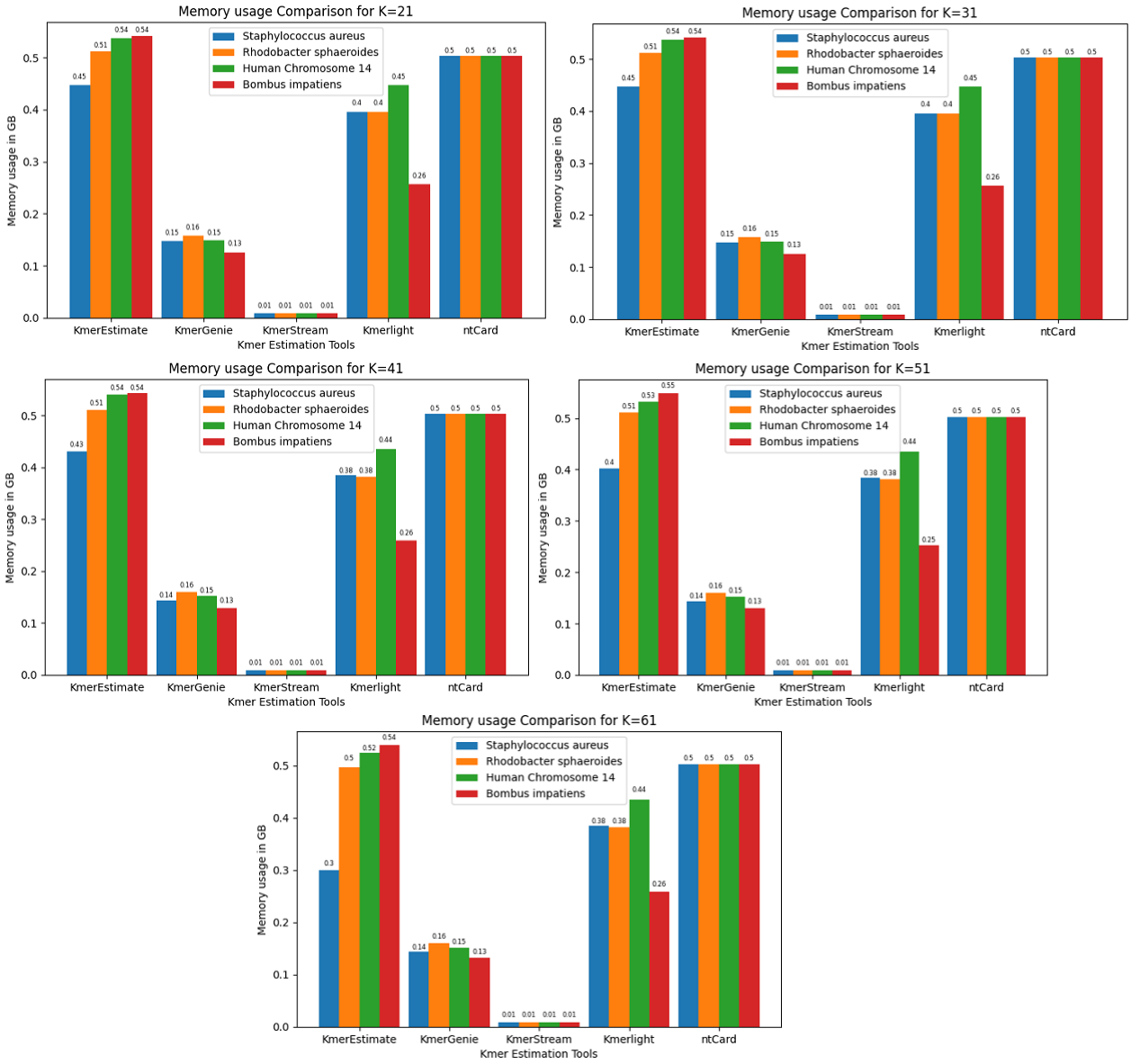

Figure 5.10: Memory Usage Comparison of Kmer Estimation Tools for different K-values

Figure 5.10shows that RAM utilization is highest for ntCard followed by kmerEstimate, kmerGenie, kmerlight and kmerStream.

Table 5.11: Number of distinct kmers (F0) and Error Rate Comparison for Human Chromosome 14

K	DSK	KmerStream	ntCard	KmerGenie	KmerEstimate	Kmerlight
21	247064189	268103932	247250177	143700000	246961360	207422975
31	281057848	275955949	281369605	136100000	281010368	181369688
41	290053061	288524815	290338792	127800000	290164928	210196367
51	283177203	281612968	283508416	119100000	283144544	243388296

Error Rate compared to that of DSK

K	KmerStream	ntCard	KmerGenie	KmerEstimate	Kmerlight
21	2.0%	0.1%	41.8%	0%	16%
31	1.8%	0.1%	51.6%	0%	35.5%
41	0.5%	0.1%	55.9%	0%	27.5%
51	0.6%	0.1%	57.9%	0%	14.1%

Table 5.12: Number of distinct kmers (F0) and Error Rate Comparison for *Bombus impatiens*

K	DSK	KmerStream	ntCard	KmerGenie	KmerEstimate	Kmerlight
21	1535193264	1508402269	1536577900	1378300000	1534383232	1272329783
31	1796695506	1738141499	1797527691	1362100000	1796788096	1214790683
41	1926334104	1880311887	1928570500	1326100000	1925727488	1421942589
51	1982316483	1954245096	1984379848	1276300000	1982516736	1400297339

Error Rate compared to that of DSK

K	KmerStream	ntCard	KmerGenie	KmerEstimate	Kmerlight
21	1.7%	0.1%	10.2%	0.1%	17.1%
31	3.3%	0%	24.2%	0%	32.4%
41	2.4%	0.1%	31.2%	0%	26.2%
51	1.4%	0.1%	35.6%	0%	29.4%

5.2.3 Discussion - Kmer Estimation Tools

5.2.3.1 Streaming Algorithms:

KmerEstimate, KmerStream, Kmerlight and ntCard are the kmer estimation algorithms based on the streaming approach. Based on figures 5.5 the execution time of Kmerstream is the best. However, tables 5.11 and 5.12 show that Kmerstream underestimates the count of distinct kmers for *Bombus impatiens* and Human Chromosome 14 datasets compared to that of our benchmark data obtained from DSK kmer counting tool for different values of k. ntCard and KmerEstimate have the

closest estimation of distinct kmers followed by by Kmerstream. KmerStream has the best RAM utilization as compared to all the other estimation tools, as shown in figures 5.10. ntCard and KmerEstimate has RAM utulization greater than that of KmerStream, which can be attributed to the frequency histogram that they provide unlike KmerStream, which provides the count of singleton, distinct and total kmers in the dataset.

5.2.3.2 Sampling Algorithms:

KmerGenie is the only estimation algorithm based on the sampling approach. KmerGenie provides abundance histogram of kmers for a range of k values. The sampling rate of the algorithm varies for different data sets depending on their size. This algorithm has the second best memory utilization as compared to the streaming algorithms as shown in figures 5.10. It has the highest run time as compared to streaming algorithms as shown in 5.5. Tables 5.11 and 5.12 show that the error rate in providing a count of distinct kmers compared to that DSK is higher than ntCard and KmerEstimate.

Chapter 6

Conclusion and Future Work

Rapid growth of DNA sequencing technologies has lead to the study of kmer counting and estimation techniques. Each of the de Bruijn graph based tools use different data structures and techniques and produce variable number of kmers. It is important to understand which tool to use given the available resources (RAM utilization and execution time). There is not a single tool that can provide the best performance interms of both memory usage and execution time. The performance depends on variable factors such as the size of the dataset and the type of dataset (whether the dataset contains a large number of unique kmers). Our analysis of the kmer counting tools shows that KMC2, Gerbil, MSPKmerCounter, DSK, Squakr and GenomeTester4 are some of the tools that made the best use of the available resources and provided results. Although KmerStream is the kmer estimation method that has the best execution time and memory usage, ntCard and KmerEstimate provided a count of distinct kmers comparable to that of DSK. As a lot of new kmer counting and estimation tools are being developed, there is a need to continue the analysis of these tools.

Bibliography

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. *The Enhanced Suffix Array and Its Applications to Genome Analysis*, pages 449–463. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[2] Wilhelm J. Ansorge. Next-generation {DNA} sequencing techniques. *New Biotechnology*, 25(4):195 – 203, 2009.

[3] Eduardo G. Dupim Antonio Bernardo Carvalho and Gabriel Goldstein. Improved assembly of noisy long reads by k-mer validation. *Published by Cold Spring Harbor Laboratory Press*, 26(1710):10, 2016.

[4] Eduardo G. Dupim Antonio Bernardo Carvalho and Gabriel Goldstein. nthash: recursive nucleotide hashing. *Bioinformatics*, 32(22):3492, 2016.

[5] Peter Audano and Fredrik Vannberg. Kanalyze: a fast versatile pipelined k-mer toolkit. *Bioinformatics*, 30(14):2070–2072, 2014.

[6] Sairam Behera, Sutanu Gayen, Jitender Deogun, and N. Vinodchandran. Kmer-estimate: A streaming algorithm for estimating k-mer counts with optimal space usage. pages 438–447, 08 2018.

[7] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[8] Rayan Chikhi and Paul Medvedev. Informed and automated k-mer size selection for genome assembly. *Bioinformatics*, 30(1):31, 2014.

[9] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58 – 75, 2005.

[10] Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. Kmc 2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, 2015.

[11] Rechner S. MÃijller-Hannemann M. Erbert, M. Gerbil: a fast and memory-efficient k-mer counter with gpu-support. *Algorithms for Molecular Biology*, 2017.

[12] Remm M. Kaplinski L, Lepamets M. Genometester4: a toolkit for performing basic set operations - union, intersection and complement on k-mer lists. *Gigascience*, 2015.

[13] Stefan Kurtz, Apurva Narechania, Joshua C. Stein, and Doreen Ware. A new method to compute k-mer frequencies and its application to annotate large repetitive plant genomes. *BMC Genomics*, 9(1):517, 2008.

[14] Edya Ladan-Mozes and Nir Shavit. *An Optimistic Approach to Lock-Free FIFO Queues*, pages 117–131. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[15] Y. Li and XifengYan. MSPKmerCounter: A Fast and Memory Efficient Approach for K-mer Counting. *ArXiv e-prints*, May 2015.

[16] Yang Li, Pegah Kamousi, Fangqiu Han, Shengqi Yang, Xifeng Yan, and Subhash Suri. Memory efficient minimum substring partitioning. *Proc. VLDB Endow.*, 6(3):169–180, January 2013.

[17] Siliang Li-Ni Hu Yimin He Ray Pong Danni Lin Lihua Lu Maggie Law Lin Liu, Yinhu Li. Comparison of next-generation sequencing systems. *Journal of Biomedicine and Biotechnology*, 2012(1):11, 2012.

[18] Abdullah-Al Mamun, Soumitra Pal, and Sanguthevar Rajasekaran. Kcmbt: a k -mer counter based on multiple burst trees. *Bioinformatics*, 32(18):2783, 2016.

[19] Swati C Manekar and Shailesh R Sathe. A benchmark study of k-mer counting methods for high-throughput sequencing. *GigaScience*, 7(12), 10 2018. giy125.

[20] Guillaume MarÃğais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

[21] Páll Melsted and Jonathan K. Pritchard. Efficient counting of k-mers in dna sequences using a bloom filter. *BMC Bioinformatics*, 12(1):333, 2011.

[22] PÃąll Melsted and Bjarni V. HalldÃşrsson. Kmerstream: Streaming algorithms for k-mer abundance estimation. *Bioinformatics*, 2014.

[23] Hamid Mohamadi, Hamza Khan, and Inanc Birol. ntcard: A streaming algorithm for cardinality estimation in genomics data. *Bioinformatics*, 2017.

[24] Pop Mihai Nagarajan Niranjan. Sequence assembly demystified. *Nat Rev Genet*, 14(157):167, 2013.

[25] Prashant Pandey, Michael Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. pages 775–787, 05 2017.

[26] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. Squeakr: an exact and approximate k-mer counting system. *Bioinformatics*, 34(4):568–575, 10 2017.

[27] Gutierrez Miguel PÃĺrez Nelson and Vera Nelson. Computational performance assessment of k-mer counting algorithms. *Journal of Computational Biolog*, 23(4), April 2016.

[28] Felix Putze, Peter Sanders, and Johannes Singler. *Cache-, Hash- and Space-Efficient Bloom Filters*, pages 108–121. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[29] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. Dsk: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652, 2013.

[30] Rajat Shuvro Roy, Debashish Bhattacharya, and Alexander Schliep. Turtle: Identifying frequent k-mers with cache-efficient algorithms. *Bioinformatics*, 30(14):1950–1957, 2014.

[31] Daniel Rubino. Ie9 for windows phone 7: Adobe flash, demos and development.

[32] Nicole Rusk. Sequence assembly demystified. *Nat Rev Genet*, 14(157):167, 2013.

[33] Bernardo Sachman-Ruiz, Verónica Narváez-Padilla, and Enrique Reynaud. Commercial bombus impatiens as reservoirs of emerging infectious diseases in central méxico. *Biological Invasions*, 17(7):2043–2053, 2015.

[34] F. Sanger and A.R. Coulson. A rapid method for determining sequences in dna by primed synthesis with dna polymerase. *Journal of Molecular Biology*, 94(3):441 – 448, 1975.

[35] Ranjan Sinha and Justin Zobel. Cache-conscious sorting of large sets of strings with dynamic tries. *J. Exp. Algorithmics*, 9, December 2004.

[36] Naveen Sivadasan, Rajgopal Srinivasan, and Kshama Goyal. Kmerlight: fast and accurate k-mer abundance estimation, 2016. arXiv:1609.05626v1.

[37] John M Urban, Jacob Bliss, Charles E Lawrence, and Susan A Gerbi. Sequencing ultra-long dna molecules with the oxford nanopore minion. *bioRxiv*, 2015.

[38] Qingpeng Zhang, Jason Pell, Rosangela Canino-Koning, Adina Chuang Howe, and C. Titus Brown. These are not the k-mers you are looking for: Efficient online k-mer counting using a probabilistic data structure. *PLOS ONE*, 9(7):1–13, 07 2014.